Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Bachelorarbeit Nr. 276

# Erweiterbares Monitoring Backend für OpenTOSCA

Tom Rohloff

| | |
|---|---|
| **Studiengang:** | Softwaretechnik |
| **Prüfer:** | Prof. Dr. Dr. h. c. Frank Leymann |
| **Betreuer:** | Dipl.-Inf. Florian Haupt |
| **Beginn am:** | 10.11.2015 |
| **Beendet am:** | 11.05.2016 |
| **CR-Nummer:** | C.2.4, H.3.2, H.3.5 |

**Kurzfassung**

Bei der Provisionierung von Cloud-Anwendungen laufen oft langwierige und komplexe Prozesse ab. Im Rahmen der TOSCA-Laufzeitumgebung OpenTOSCA wurde in einem Vorprojekt ein System entwickelt, um während dieser Prozesse Ereignisse zu erfassen und abrufbar zu machen. Dieses Monitoring Backend hat jedoch einige Schwachstellen, welche die Wartbarkeit und Erweiterbarkeit erschweren.

In dieser Arbeit werden die Anforderungen an ein generisches, erweiterbares Monitoring Backend identifiziert, ein Entwurf für eine REST-Schnittstelle sowie die Speicherung ausgearbeitet, dieser Entwurf implementiert und letztlich eine Erweiterung für die OpenTOSCA-Laufzeitumgebung entworfen und implementiert.

# Inhaltsverzeichnis


**1 Einleitung** **9**
1.1 Ziel der Bachelorarbeit . . . . . . . . . . 10
1.2 Aufbau der Bachelorarbeit . . . . . . . . . . 10

**2 Grundlagen** **11**
2.1 REST . . . . . . . . . . 11
2.1.1 Einschränkungen (Constraints) von REST . . . . . . . . . . 11
2.2 HTTP . . . . . . . . . . 15
2.2.1 Operationen . . . . . . . . . . 16
2.2.2 Anfragen (Requests) und Antworten (Responses) . . . . . . . . . . 18
2.2.3 HTTP und REST . . . . . . . . . . 20
2.2.4 Arten von Ressourcen . . . . . . . . . . 20
2.2.5 Richardson Maturity Model . . . . . . . . . . 21
2.3 TOSCA . . . . . . . . . . 22
2.3.1 Topology . . . . . . . . . . 22
2.3.2 Orchestration . . . . . . . . . . 25
2.3.3 CSAR . . . . . . . . . . 25
2.3.4 OpenTOSCA . . . . . . . . . . 26

**3 Bisherige Lösung** **27**
3.1 Aufbau und Funktion . . . . . . . . . . 27
3.2 Schwachstellen . . . . . . . . . . 28

**4 Entwurf des Monitoring Backends** **31**
4.1 Entwurf der REST-API . . . . . . . . . . 31
4.1.1 Anforderungen . . . . . . . . . . 31
4.2 Umsetzung des API-Entwurfs . . . . . . . . . . 32
4.2.1 Use Cases . . . . . . . . . . 35
4.2.2 Mechanismen bei der API-Interaktion . . . . . . . . . . 36
4.2.3 URL-Struktur der REST-API . . . . . . . . . . 37
4.3 Entwurf des Grundsystems . . . . . . . . . . 37
4.3.1 Architektur . . . . . . . . . . 38
4.3.2 Funktionsweise . . . . . . . . . . 39
4.3.3 Ableitung / Vererbung . . . . . . . . . . 41
4.4 Entwurf einer Erweiterung: OpenTOSCA . . . . . . . . . . 42
4.4.1 REST-API . . . . . . . . . . 42
4.4.2 OpenTOSCA-Komponenten . . . . . . . . . . 45

**5 Implementierung** **47**
5.1 Genutzte Technologien und Abhängigkeiten . . . . . . . . . . 47
5.1.1 Versionskontrolle . . . . . . . . . . 47
5.1.2 Entwicklungsumgebung . . . . . . . . . . 47
5.1.3 Maven . . . . . . . . . . 48

5.1.4 Tomcat . . . 48
5.1.5 JAX-RS / Jersey . . . 48
5.1.6 Java Persistence API / Eclipselink / Derby . . . 48
5.1.7 SoapUI / Tests . . . 49
5.1.8 Swagger . . . 49
5.2 REST-API . . . 49
5.3 Persistenz . . . 50
5.4 Tests . . . 52

**6 Zusammenfassung und Ausblick 54**

# Abbildungsverzeichnis


2.1 Herleitung von REST durch Style Constraints (nach [1]) . . . . . . . 12
2.2 Charakteristiken von (RESTful-)Sessions (nach [4]) . . . . . . . . . . 14
2.3 HTTP im OSI-Schichtenmodell der Netzwerkprotokolle (nach [3]) . . 15
2.4 Weiterleitung durch Ressourcen (nach [4]) . . . . . . . . . . . . . . . 21
2.5 Ein TOSCA Topology Template und seine Komponenten . . . . . . . 23
2.6 Beispiel einer TOSCA Topology für einen Webshop . . . . . . . . . . 24
2.7 Inhalt eines Cloud Service Archivs (nach [17]) . . . . . . . . . . . . . 26
3.1 Zusammenhang zwischen den Komponenten von Monitor Server und OpenTOSCA (bisherige Lösung) . . . . . . . . . . . . . . . . . . . . . 27
3.2 System mit geringer Kopplung und hohem Zusammenhalt (Grafik aus [18]) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.1 Aufgaben der REST-API des Monitoring Backends . . . . . . . . . . 32
4.2 REST-Schnittstelle für den *Basic-Storage* . . . . . . . . . . . . . . . 34
4.3 Use Cases aus Sicht von Frontend und Eventquelle . . . . . . . . . . . 35
4.4 Architektur des Grundsystems (Basic-Storage) . . . . . . . . . . . . . 38
4.5 Speichern und Löschen eines Events über die REST-API . . . . . . . 39
4.6 Kapselung der Systemkomponenten . . . . . . . . . . . . . . . . . . . 40
4.7 REST-Schnittstelle für die OpenTOSCA-Erweiterung . . . . . . . . . 43
4.8 REST-Schnittstelle für den Basic Storage (links) und OpenTOSCA-Erweiterung (rechts) . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
4.9 Zusätzliche Use Cases bei OpenTOSCA . . . . . . . . . . . . . . . . . 45
4.10 Vererbung des Datenmodells an die OpenTOSCA-Erweiterung . . . . 46
5.1 Speicherung der Events: Interface und Implementierung des DAOs . . 51
5.2 Testfall: Events abfragen und Event speichern . . . . . . . . . . . . . 53


# Listings


1 HTTP-Request Schema . . . . . . . . . . . . . . . . . . . . . . . . . 18
2 Einfacher HTTP-Request . . . . . . . . . . . . . . . . . . . . . . . . 18
3 Einfache HTTP-Response . . . . . . . . . . . . . . . . . . . . . . . . 19
4 Event-Daten im Message-Body . . . . . . . . . . . . . . . . . . . . . 36
5 Event-Daten als Query-Parameter . . . . . . . . . . . . . . . . . . . . 36
6 HTTP-Request mit OpenTOSCA-Event als Nutzlast . . . . . . . . . 42
7 JAX-RS Annotationen bei der REST-API . . . . . . . . . . . . . . . 50
8 JPA-Annotationen zur persistenten Speicherung von Events . . . . . 52

# 1 Einleitung

Mittlerweile durchdringen Computer die meisten Bereiche eines Unternehmens. Doch nicht nur die Hardware tritt in vielerlei Facetten an immer mehr Stellen auf, auch die Software gewinnt immer größere Bedeutung. Entgegen der früher üblichen Nutzung reiner Terminal- oder Desktop-Anwendungen werden jetzt immer mehr Anwendungen über Web-Schnittstellen definiert. Dies bietet viele Vorteile sowohl bei der Entwicklung, Verwaltung (Administration), als auch bei der Nutzung der Anwendung. Das Hauptaugenmerk bei der Entwicklung liegt dabei auf der Wiederverwendbarkeit und Erweiterbarkeit von Komponenten. Erreicht werden diese Eigenschaften unter Anderem durch eine geringe Kopplung, einen hohen Zusammenhalt und eine gute Dokumentation.
Solche Web-Schnittstellen werden oftmals mit Hilfe von Representational State Transfers (REST) geschaffen. Dieses von Roy Fielding [1] eingeführte Paradigma ermöglicht eine von konkreten Implementierungsdetails abstrahierte Sichtweise auf die an der Kommunikation beteiligten Komponenten.

Große Aufmerksamkeit hat in der jüngst vergangenen Zeit das sogenannte Cloud-Computing erfahren. Dieser „on-demand"-Ansatz ermöglicht es, Ressourcen wie etwa Netzwerke, Datenspeicher oder ganze Service-Anwendungen mit relativ geringem Verwaltungs- und Konfigurationsaufwand zur Verfügung zu stellen und ebenso schnell wieder freizugeben. Für Nutzer solcher Cloud-Dienste können sich damit verschiedene Vorteile ergeben wie etwa die Konzentration auf das dahinterstehende Projekt anstatt auf seine Infrastruktur.

Betrachtet man das *Software as a Service (SaaS)* Service-Modell von Cloud-Anwendungen, so liegt der Fokus nicht auf der zur Verfügung gestellten Infrastruktur, sondern auf einer bestimmten Anwendungssoftware. Eine solche Cloud-Anwendung besteht üblicherweise aus mehreren Komponenten wie zum Beispiel einer Weboberfläche für den Zugang des Endnutzers und einer Datenbank im Hintergrund, die für die Speicherung benötigt wird. Die Provisionierung (Ausbringung, Deployment) dieser Anwendungen sind oft Prozesse, die lange laufen und eine hohe Komplexität aufweisen.

An der Universität Stuttgart wurde in den letzten Jahren eine Laufzeitumgebung für Cloud-Anwendungen entwickelt, die auf dem TOSCA-Standard aufbauen. Dieses OpenTOSCA genannte Projekt soll nun an verschiedenen Punkten erweitert und weiterentwickelt werden.

Um die Prozesse bei der Provisionierung für den Nutzer nachvollziehbar und -verfolgbar zu machen, wurde in einem Vorprojekt ein System für OpenTOSCA entwickelt, das einzelne Ereignisse während der Provisionierung erfasst und zur späteren Verwendung zugreifbar macht. Dieses System hat jedoch einige Schwachstellen, woraus der Bedarf nach einer überarbeiteten Fassung entstand. Im Rahmen dieser Bachelorarbeit wurde ein Monitoring-Backend entwickelt, das die

Schwierigkeiten des alten Systems überwindet und den Anforderungen nach Erweiterbarkeit und REST-Konformität gerecht wird.

## 1.1 Ziel der Bachelorarbeit

Zu den Zielen dieser Bachelorarbeit gehört:

- Identifikation der Anforderungen an ein erweiterbares Monitoring Backend
- Entwurf der REST API des Dienstes
- Implementierung der REST API
- Entwurf einer beispielhaften Erweiterung des Monitoring Backend
- Implementierung einer beispielhaften Erweiterung des Monitoring Backend

## 1.2 Aufbau der Bachelorarbeit

Der Rest dieser Bachelorarbeit gliedert sich wie folgt: Um einen Überblick über das Themengebiet REST und OpenTOSCA zu bekommen, wird im zweiten Kapitel auf die Grundlagen eingegangen. In Kapitel 3 wird das Vorgängerprojekt vorgestellt, seine Funktionsweise erläutert und vorhandene Schwachstellen identifiziert. Kapitel 4 beschäftigt sich mit dem Entwurf des Monitoring Backends, welcher unterteilt ist in mehrere Teile. So wird zunächst der Entwurf der REST-API beschrieben, indem die Anforderungen ermittelt werden und die Schnittstelle ausformuliert wird. Anschließend werden die Komponenten hinter der REST-API behandelt, die sich um die Weiterverarbeitung und Speicherung der Daten kümmern. Das fünfte Kapitel widmet sich der Implementierung des zuvor entwickelten Entwurfes. Hier werden die verwendete Technologien kurz vorgestellt und Schlüsselelemente bei der Programmierung der REST-Schnittstelle und der Speicherung näher beleuchtet. Das sechste und letzte Kapitel beinhaltet eine Zusammenfassung dieser Bachelorarbeit und gibt einen Ausblick über mögliche Erweiterungen des Monitoring Backends.

# 2 Grundlagen

In diesem Abschnitt wird auf einige Grundlagen eingegangen, welche ein leichteres Verständnis der entwickelten API ermöglichen. Zunächst wird hierzu REST als Architekturstil behandelt, anschließend die sehr verbreitete Umsetzung mit HTTP. Abschließend wird auf den TOSCA-Standard eingegangen.

## 2.1 REST

REST (Representational State Transfer) bezeichnet einen von Roy Fielding [1] erarbeiteten Architekturstil zur Entwicklung von verteilten Anwendungen. Hauptmerkmal ist die Konzentration auf die Rollen einzelner Komponenten einer Anwendung, weniger auf deren Implementierungsdetails oder Kommunikationsprotokolldetails.

### 2.1.1 Einschränkungen (Constraints) von REST

Fieldung definiert REST als eine Menge von Einschränkungen, der er iterativ weitere *Constraints* hinzufügt, um so Stück für Stück einen hybriden Architekturstil abzuleiten.

Dazu beginnt er mit der leeren Menge, dem sogenannten *Null style*. Dieser entspricht einem System, dem noch keine speziellen Einschränkungen auferlegt sind.

Diesem Stil wird als erste Einschränkung der *Client-Server*-Architekturstil hinzugefügt. Die Aufteilung in einen Client, der Anfragen schickt, und einen Server, der darauf antwortet, stellt eine Trennung von Zuständigkeiten dar. Dadurch ist es möglich, dass Komponenten unabhängig von einander entwickelt werden können. Ferner erhöht diese Trennung die Portabilität der Benutzerschnittstelle über verschiedene Plattformen hinweg und verbessert die Skalierbarkeit, indem die Serverkomponenten vereinfacht werden.

Die zweite Einschränkung ist die Forderung nach *Zustandslosigkeit (stateless)* der Kommunikation bei der Interaktion zwischen Client und Server. Dies bedeutet, dass jede Anfrage des Clients alle nötigen Informationen enthalten muss, sodass der Server die Anfrage vollständig verstehen und bearbeiten kann. Im Besonderen darf kein auf dem Server gespeicherter Kontext genutzt werden. Diese Einschränkung betrifft verschiedene Eigenschaften, die verbessert werden. Darunter die

**Sichtbarkeit,** da etwa bei einer Analyse des Datenverkehrs nur eine einzelne Nachricht betrachtet werden muss, um den Umfang der ganzen Anfrage erfassen zu können.

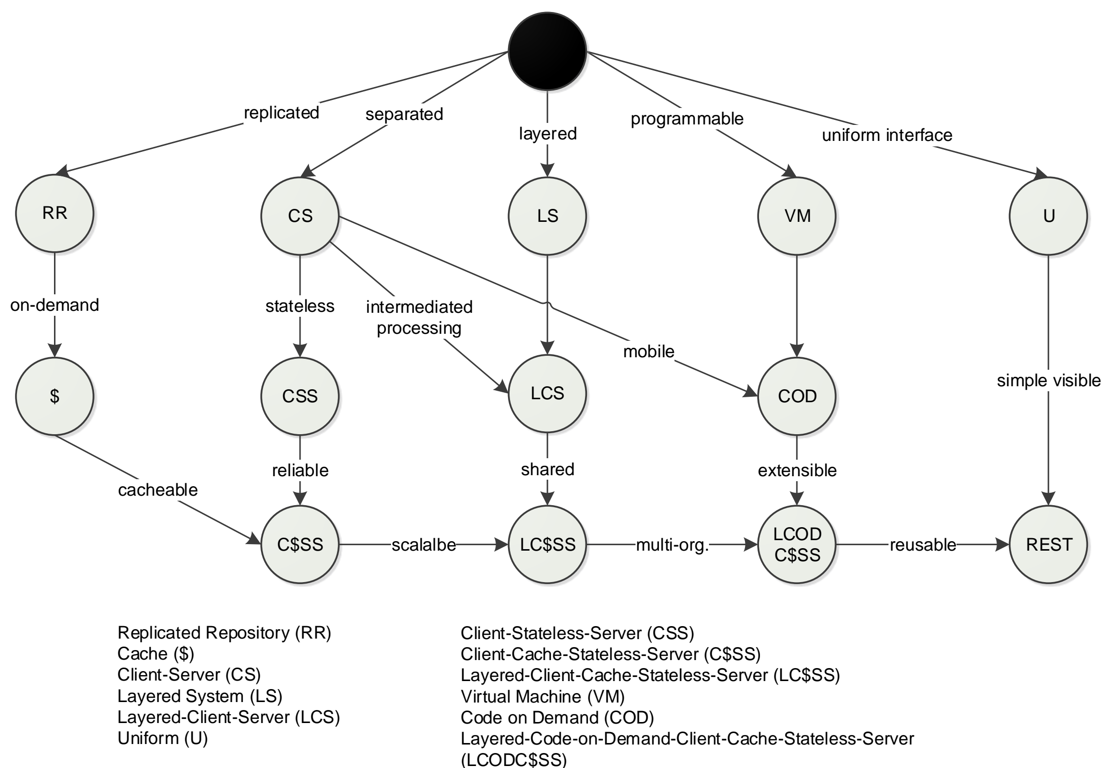


Abbildung 2.1: Herleitung von REST durch Style Constraints (nach [1])

**Zuverlässigkeit,** weil das Zurückkehren zur normalen Funktion (recovering) bei einem Fehler vereinfacht wird

**Skalierbarkeit,** da der Server keinen Zustand über mehrere Anfragen hinweg verwalten muss. Dadurch können Ressourcen wie Speicher schneller wieder freigegeben werden. Auch die für den Client transparente Verteilung auf mehrere Server wird hierdurch vereinfacht.

Ein Nachteil besteht in dem Overhead, der im Besonderen bei aufeinanderfolgenden Anfragen auftritt.
Angemerkt sei an dieser Stelle, dass die Zustandslosigkeit sich nur au die Kommunikation bezieht, nicht jedoch auf eine Sitzung (*session*). Eine Anwendung, die aus mehreren Anfragen eines Clients besteht, kann sehr wohl einen Zustand inne haben. Dessen Verwaltung obliegt jedoch vollständig dem Client.

Um den Nachteil der durch Overhead verringerten (Netzwerk-)Performanz auszugleichen, fügt Fielding als nächste Einschränkung das Konzept eines *Caches* hinzu. Dies bedeutet, dass Daten einer Antwort des Server implizit oder explizit als „*cacheable*“ beziehungsweise „*non-cacheable*“ gekennzeichnet werden. Ist eine Information cachable, so kann der Client diese Information speichern und muss sie in späteren, identischen Anfragen nicht nochmals zusätzlich erfragen. Dadurch wird die Skalierbarkeit und (Netzwerk-)Effizienz verbessert.

Als weiterer Constraint wird eine *einheitliche Schnittstelle (uniform interface)* hinzugefügt. Dieses Interface ist das Schlüsselkonzept, das REST von anderen Architekturstilen abgrenzt. Zu einer einheitlichen Schnittstelle gehört:

**Identifikation der Ressourcen:** Eine Ressource ist bei REST die Abstraktion einer Information. Sie bezeichnet alles, das wichtig genug ist, um als eigene Entität referenziert zu werden [15]. Unerheblich ist dabei, ob es sich um konkrete, leicht messbare Dinge handelt, oder um abstrakte Konzepte. Einige Beispiele sind:

- die jüngste Version einer Software
- der momentane Zustand einer Applikation
- bisher aufgetretene Fehler in einem Prozess
- die Beziehung zwischen Alice und Bob

Ressourcen werden in einer Anfrage durch einen Ressourcenbezeichner identifiziert.

**Manipulation von Ressourcen durch Repräsentationen:** Eine Repräsentation ist eine Beschreibung des momentanen Zustandes einer Ressource. Solche Repräsentationen entsprechen etwa dem Format einer bestimmten Information. Diese Unterscheidung von Darstellung und eigentlichem Inhalt ermöglicht das Anbieten mehrerer Repräsentationen einer Ressource. So könnte eine Liste von Elementen zum Beispiel in den Formaten JSON, XML und CSV repräsentiert werden. Der eigentliche Inhalt bleibt derselbe, lediglich die Art der Bereitstellung ist unterschiedlich. Die Interaktion beziehungsweise Manipulation einer Ressource erfolgt immer durch eine ihrer Repräsentationen. Welche das letztlich ist, kann zum Beispiel durch *Content Negotiation* zwischen Client und Server ausgehandelt werden.

**Selbstbeschreibende Nachrichten:** Jede Nachricht enthält genug Informationen, um zu beschreiben, wie die jeweilige Nachricht zu verarbeiten ist. Dies kann zum Beispiel über MIME[1]-Typen realisiert werden.

**HATEOAS:** HATEOAS steht für *Hypermedia as the engine of application state* und bezeichnet die Verbindung mehrerer Ressourcen untereinander durch Referenzen, welche vom Server verwaltet werden. Solche Referenzen können zum Beispiel der Repräsentation einer Ressource angefügt werden, sodass sich ein Client anhand dieser Verlinkungen von Ressource zu Ressource „hangeln“ kann.
Diese HATEOAS-Einschränkung entkoppelt Client und Server so, dass sich die Server-Funktionalität eigenständig weiterentwickeln kann und der Client

[1]Multipurpose Internet Mail Extensions, ein Standard, der das Mail-Format erweitert. Unter anderem kann dadurch ein Content-Type angegeben werden. Näheres siehe in [10] und [11].

diese Funktionalität auch nach einer Änderung noch immer nutzen kann.

Eine einheitliche Schnittstelle führt letztlich zu einer vereinfachten Systemarchitektur. Ein Nachteil dieser Einschränkung ist eine verringerte Effizienz gegenüber einer an die Anwendung angepasste, spezifische Schnittstelle.

Der Menge der Einschränkungen fügt Fielding ferner eine *Unterteilung in Schichten (layered system)* hinzu. Ein System wird dabei in hierarchisch angeordnete Schichten unterteilt, welche nur mit dem unmittelbaren Nachbarn kommunizieren können. Dadurch ist die Kapselung einzelner Komponenten leichter möglich, was jedoch Nachteile wie Overheads und Latenzerhöhungen beim der Kommunikation zwischen den Schichten mit sich bringt. Abhilfe schafft das zuvor bereits hinzugefügte Caching.

Die letzte Einschränkung bei REST ist *code on demand.* Dieser explizit als optional gekennzeichnete Constraint besagt, dass man die Funktionalität eines Clients erweitern kann, indem der Server Code etwa in Form von Applets oder Scripts zur Verfügung stellt. Diese Erweiterungen werden vom Client heruntergeladen und von ihm ausgeführt.

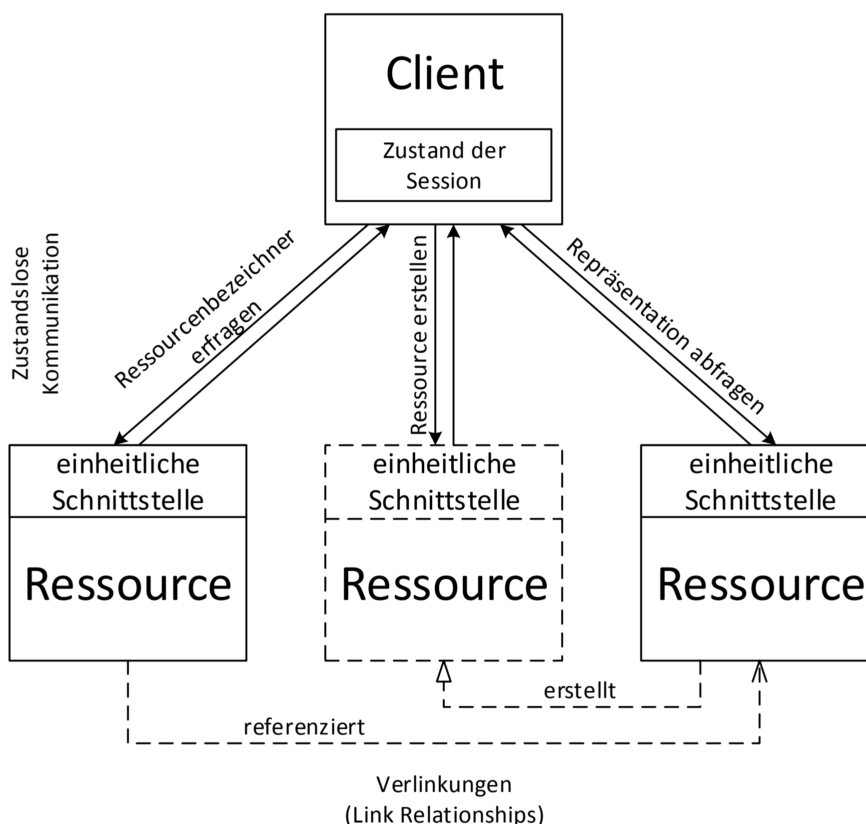


Abbildung 2.2: Charakteristiken von (RESTful-)Sessions (nach [4])

Zusammengefasst bringt das Entwickeln nach dem REST-Architekturstil mehrere Vorteile mit sich, darunter eine erhöhte Skalierbarkeit, Performanz, eine lose Kopplung, sowie eine leichtere Wiederverwendung der Komponenten. Systeme, welche die oben genannten Einschränkungen einhalten, bezeichnet man als RESTful [15] („REST folgend").
In Abbildung 2.2 sind die oben erläuterten Eigenschaften beispielhaft abgebildet. Eine Sitzung (Session) kann zum Beispiel genutzt werden, um eine Ressource nachzuschlagen (lookup), eine Repräsentation zu erfragen oder um eine neue Ressource anzulegen. Die Kommunikation verläuft dabei zustandslos, die eigentliche Sitzung (Session, auch *Konversation*) kann dabei jedoch einen Zustand haben, der vom Client verwaltet wird.

## 2.2 HTTP

HTTP ist ein Kommunikationsprotokoll, welches im OSI-Schichtenmodell in der Anwendungsschicht verortet wird [6].

Grundsätzlich werden zwei Kommunikationspartner unterschieden: Der Client stellt einen sogenannten *Request* (Anfrage) an den Server, welcher anschließend eine *Response* (Antwort) zurückgibt. Ein solcher Request enthält die Operation, welche auf die ebenfalls enthaltene Ressource angewandt wird.

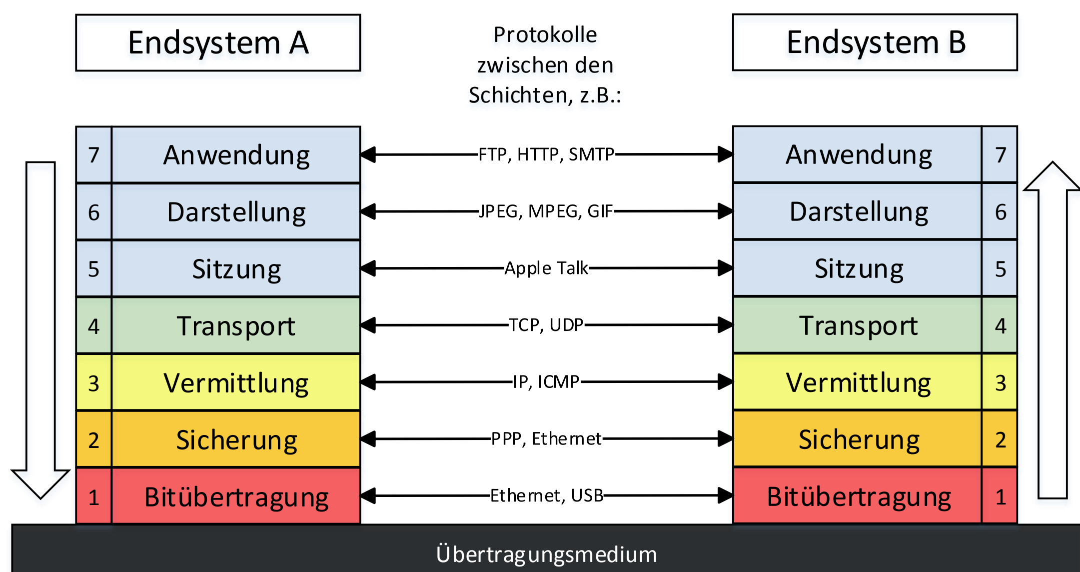


Abbildung 2.3: HTTP im OSI-Schichtenmodell der Netzwerkprotokolle (nach [3])

### 2.2.1 Operationen

Eine Operation bezeichnet die Art, wie auf eine Ressource zugegriffen wird. Sie gehört zu dem vom Client geschickten Request. Es werden drei Eigenschaften von Operationen definiert. Dies sind im Detail:

**Idempotenz (idempotent methods):** Bei idempotenten Operationen verursacht das mehrfache Ausführen derselben Operation auf einer Ressource keine weiteren Änderungen. Dies bedeutet, dass (bei gleichbleibender Operation und gleichbleibendem Ressourcenbezeichner) sich der Zustand der Ressource nur höchstens einmal ändert. Alle Ausführungen nach der ersten bewirken keine erneute Änderung.

**Sicherheit (safe methods):** Die Sicherheit einer Operation ist gegeben, wenn keine Seiteneffekte bei der Ausführung auftreten. Im Besonderen wird der Zustand der Ressource nicht geändert. In der Praxis werden (kleine) Seiteneffekte, wie sie etwa durch das Logging des Servers auftreten, ignoriert. Eine sichere Operation ist immer auch idempotent, da das mehrmalige Ausführen zu keiner Änderung führt.

**Zwischenspeicherbarkeit (cacheable methods):** Eine Operation wird „zwischenspeicherbar“ genannt, wenn die Antwort auf einen Request für zukünftige Verwendung zwischengespeichert werden kann. Näheres zu Caching bei HTTP entnimmt man [7].

Nachfolgend werden die möglichen Operationen näher beschrieben. Tiefergehende Details hierzu findet man unter [6].

**GET:** Mit GET wird eine Repräsentation der Ressource erfragt. Es entspricht dem einfachen, lesenden Zugriff auf eine bestimmte Information, die durch eine Ressource bereitgestellt wird. Mit vermuteten 95% [16] macht diese Operation den größten Anteil der im World Wide Web verwendeten HTTP-Verben aus. Sie soll serverseitig keine weiteren Auswirkungen haben. Daraus ergibt sich die Forderung, dass GET safe (und damit idempotent) sein sollte.

**POST:** Mit POST werden Daten an den Server geschickt, die dieser anschließend verarbeitet. Grundsätzlich wird der Server aufgefordert, die mitgeschickten Informationen als der Ressource untergeordnet zu akzeptieren. Dies bedeutet allerdings nicht automatisch, dass eine neue (Sub-)Ressource angelegt wird.
Typische Anwednung findet POST etwa beim Erweitern einer Datenbank (z.B. bei einem Forum) oder dem Übertragen eines Datenblocks zur weiteren Verarbeitung (etwa bei einem HTML-Form-Element). POST ist weder idempotent, noch safe. Das mehrmalige Operieren auf einer Ressource mit POST könnte beispielsweise mehrere Ressourcen mit gleichem Inhalt anlegen.

**PUT:** Mit PUT wird der Server aufgefordert, die im Anfragekörper enthaltene Repräsentation als Ressource unter dem angegebenen Ressourcenbezeichner zur Verfügung zu stellen. Ist dieser Bezeichner bereits vorhanden, so wird die dahinterstehende Ressource ersetzt. Da mehrmaliges Ausführen zu keiner weiteren Änderung führt, ist PUT zwar nicht safe, aber idempotent. Der Hauptunterschied zu POST liegt in der Bedeutung des Ressourcenbezeichners bei den beiden Operationen. Bei POST identifiziert der Bezeichner die Ressource, welche die Daten des Anfragekörpers bearbeitet. Bei PUT hingegen identifiziert der Ressourcenbezeichner die im Anfragekörper enthaltene Entität (bzw. Repräsentation der Ressource) selbst. Ein Server darf eine PUT-Operation nicht auf eine andere als die durch den Bezeichner angegebene Ressource anwenden.

**DELETE:** Mit DELETE wird die durch den Ressourcenbezeichner identifizierte Ressource gelöscht. Offenbar ist DELETE nicht safe, allerdings sollte diese Operation idempotent sein: Das mehrmalige Löschen derselben Ressource sollte nicht zu weiteren Änderungen über das erste Entfernen hinaus führen.

**HEAD:** Die HEAD-Operation ist sehr ähnlich zu GET, mit dem entscheidenden Unterschied, dass der Antwortkörper leer sein muss, die Header-Felder hingegen dieselben wie bei GET sein sollten. Diese Operation wird oft dazu verwendet, um die Existenz einer Ressource zu überprüfen. Entsprechend ist sie idempotent und safe.

**OPTIONS:** Mit OPTIONS können die auf einer Ressource möglichen Operationen erfragt werden, ohne die jeweilige Operation dabei auszuführen. Dies ist idempotent und safe.

**TRACE:** Bei TRACE bekommt der Client als Antwort seine eigene Anfrage in exakt der Form, wie sie beim Server angekommen ist. Diese Antwort inklusive der Header befindet sich im Antwortkörper, der MIME-Typ ist hierbei *message/http*.

**CONNECT:** CONNECT wird von Proxys genutzt, die in der Lage sind, einen SSL-Tunnel anzubieten.

**PATCH:** Die PATCH-Operation entstand aus dem Wunsch, nur Teile einer Ressource zu ändern, statt mit PUT die gesamte Repräsentation einer Ressource zu ersetzen [12]. Dies ist besonders bei großen Ressourcen-Repräsentationen und kleinen Änderungen sinnvoll, um die genutzte Bandbreite auf ein Minimum zu reduzieren. Da PATCH etwa zehn Jahre jünger als HTTP 1.1 ist, kann die Unterstützung in der Praxis noch als entsprechend gering angesehen werden.

### 2.2.2 Anfragen (Requests) und Antworten (Responses)

Eine HTTP-Anfrage besteht aus mehreren Textzeilen und hat die folgende, grundsätzliche Form:

- eine Anfragezeile
- Host-Header
- weitere, optionale Request-Header-Zeilen
- eine Leerzeile
- optionaler Inhalt (Request Body)

**Listing 1** HTTP-Request Schema

```
1 OPERATION RESSOURCE HTTP/1.1
2 Host: hostname
```

Die Anfragezeile besteht aus genau einer Operation, dem Ressourcenbezeichner der angefragten Ressource, sowie der verwendeten HTTP-Version.
Eine Header-Zeile ist ein durch einen Doppelpunkt getrenntes Schlüssel-Wert-Paar. Seit HTTP Version 1.1 ist das Host-Feld verpflichtend [8], es gibt also mindestens eine Header-Zeile. Über die Header-Felder können unterschiedliche Repräsentationen einer Ressource erfragt oder geschickt werden, indem zum Beispiel MIME-Typen angegeben werden.

**Listing 2** Einfacher HTTP-Request

```
1 GET /index.html HTTP/1.1
2 Host: www.opentosca.org
```

Im obenstehenden Listing 2 wird eine Anfrage an den Host `www.opentosca.org` gestellt. Der Ressourcenbezeichner ist `/index.html`, die verwendete Operation `GET`, das genutzte Protokoll ist `HTTP/1.1`. Der vollständige Pfad der Ressource (URL[1]) lautet damit also: `http://www.opentosca.org/index.html`
Hat der Server die Anfrage empfangen und verarbeitet, antwortet er mit einer (mehrzeiligen) Response, die ebenfalls einem bestimmten Schema entspricht:

- eine Statuszeile
- optionale Response-Header-Zeilen
- eine Leerzeile

[1] Uniform Resource Locator. Ein URL bezeichnet eine Untermenge von URIs (Uniform Resource Identifier), bei welcher zusätzlich zur Identifikation noch die Zugriffsmechanismen (auf den Ort im Netzwerk, daher Locator) beschrieben werden. Näheres hierzu siehe unter [13].

- optionaler Inhalt (Response Body)

Eine mögliche Antwort auf den obenstehenden Request könnte etwa wie folgt aussehen:

**Listing 3** Einfache HTTP-Response

```
HTTP/1.1 200 OK
Date: Mon, 11 Apr 2016 14:31:30 CET
Content-Type: text/html; charset=UTF-8
Content-Length: 147

<html>
<head>
<title>This is the index.html Resource</title>
</head>
<body>
Current representation is text/html, encoded with UTF-8
</body>
</html>
```

In der Statuszeile findet man die Protokollversion, sowie den HTTP-Statuscode (im Listing ist dieser 200). Über die entsprechenden Header-Felder können Metainformationen mitgeschickt werden, die den eigentlichen Inhalt weiter beschreiben. Häufig findet man hier die Darstellungsform (Repräsentation), einen Zeitstempel oder die Länge der Antwort.

**HTTP-Statuscodes** werden vom Server genutzt, um dem Client Informationen über den Status der Verarbeitung seiner Anfrage vermitteln zu können. Sie werden auf jeden HTTP-Request als Antwort zurückgeschickt. Es gibt fünf Klassen [9], welche eine grobe Einteilung der Codes vorgeben:

- 1xx - Informational: Informationen zur Bearbeitung der aktuellen Anfrage
- 2xx - Successful: Die Anfrage wurde bearbeitet, die Response kann verwendet werden
- 3xx - Redirection: Weiterleitungen von der angefragten Ressource zu einer anderen
- 4xx - Client Error: Die Anfrage konnte nicht beantwortet werden, der Fehler liegt beim Client
- 5xx - Server Error: Die Anfrage konnte nicht beantwortet werden, der Fehler liegt beim Server

### 2.2.3 HTTP und REST

REST bezeichnet einen Architekturstil für verteilte Systeme, HTTP hingegen eine Möglichkeit der Implementierung. Bestimmte REST-Constraints werden bei der Verwendung von HTTP implizit befolgt. So ist bei HTTP zwingend ein Client und ein Server vorgeschrieben. Ob andere Constraints dagegen eingehalten werden, hängt von der Implementierung der jeweiligen Anwendung ab. Dies betrifft zum Beispiel die weiter oben beschriebene Zustandslosigkeit der Kommunikation. Zu unterscheiden ist deshalb auch eine RESTful- API von einer HTTP API. Ersteres bezeichnet eine Schnittstelle, die die REST-Constraints einhält. Zweiteres ist jede Schnittstelle, die HTTP als Kommunikationsprotokoll nutzt. Um ein Maß zu haben, wie sehr eine HTTP API sich an die REST-Constraints halt, wurde das in Abschnitt 2.2.5 erläuterte Richardson Maturity Model eingeführt.

### 2.2.4 Arten von Ressourcen

Von Tilkov [16] werden verschiedene Arten von Ressourcen unterschieden. Die für diese Arbeit wichtigen sind die Primärressourcen, Subressourcen und Listenressourcen:

**Primärressourcen** bezeichnen die wichtigsten Entitäten einer Anwendung. Durch die Transparenz kann der Nutzer einer Primärressource keine Rückschlüsse auf die Implementierung ziehen. Mögliche Primärressourcen wären bei einem Webshop zum Beispiel:

- http://webshop.tld/impressum
- http://webshop.tld/about

**Subressourcen** sind Teile einer (größeren) Ressource, die für sich gesehen bereits den Umfang einer eigenen Ressource erreicht haben. Ein klassisches Beispiel wären die vorhandenen Einträge einer Datenbank - im Gegensatz zur gesamten Datenbank:

- http://example.com/books/examplebook/first_chapter

**Listenressourcen** (auch *collection ressources* genannt [4]) sind eine Zusammenführung mehrerer Primärressourcen. Wie die Primärressourcen auch, können sie verschiedene Repräsentationen haben. Eine Besonderheit stellt bei ihnen der Zugriff dar: Mit der Operation GET wird eine Repräsentation der Liste zurückgegeben, mit POST hingegen kann ein neues Element der Liste hinzugefügt werden, wobei dann die entsprechende Primärressource angelegt wird. Selbstbezeichnende Ressourcennamen erhöhen hierbei die Les- und damit Wartbarkeit stark:

- http://webshop.tld/artikel
  Obwohl nicht falsch, fällt es schwer zu unterscheiden, ob es sich um eine Listenressource handelt und ob es Subressourcen gibt oder nicht. Besser wäre hierbei die folgende Lösung:

- http://webshop.tld/artikelliste

Listenressourcen können auch eingesetzt werden, um als eine Art Verzeichnis zu dienen. Dabei kommt es zu einer Weiterleitung an die eigentliche Zielressource, wie in Abbildung 2.4 dargestellt. Der Client fragt zunächst die Ressource R1 an, verarbeitet die Antwort und erhält dabei den Ressourcenbezeichner von Ressource R2. Diese wird abschließend angefragt. Solche Verzeichnisse werden auch *home document* [4] genannt, da sie oft den Wurzelknoten in des Verzeichnisbaumes darstellen.

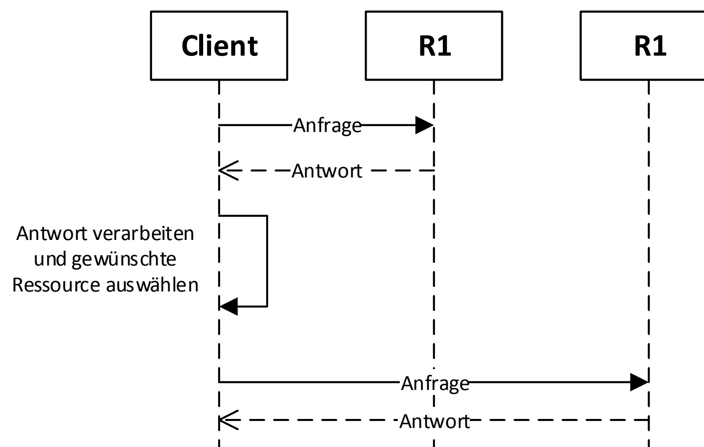


Abbildung 2.4: Weiterleitung durch Ressourcen (nach [4])

### 2.2.5 Richardson Maturity Model

Um beurteilen zu können, wie stark sich ein System an die REST-Prinzipien hält, hat Leonard Richardson das (Richardson) REST Maturity Model [2] erarbeitet. Dabei werden vier Stufen unterschieden. Das Erreichen einer Stufe (Level) erfordert das Erfüllen verschiedener REST-Prinzipien. Dabei schließt eine Stufe die vorigen mit ein, es ist also ein hierarchischer Aufbau. Im Folgenden eine Erklärung der einzelnen Stufen:

**Level 0: RESTles** Anwendungen auf diesem Level benutzen HTTP als Transportsystem für entfernte Interaktionen. Der jeweilige Dienst wird über einen einzigen Endpunkt (Ressource) bereitgestellt. Auf diesen kann

wiederum mit nur einer HTTP-Operation (meistens POST) zugegriffen werden. Es werden keine anderen Mechanismen wie etwa Verlinkungen oder unterschiedliche Operationen unterstützt. Letztlich wird HTTP dadurch als Tunnel-Mechanismus für eine eigene RPC-Variante (Remove Procedure Call) genutzt.

**Level 1: Ressourcen** Auf dieser Stufen werden mehrere Ressourcen statt eines einzige Endpunktes angeboten. Dies erlaubt bereits eine flexiblere Gestaltung eines Dienstes und der dazugehörigen Kommunikation. Noch immer wird nur mit einer HTTP-Operation auf die jeweilige Ressource zugegriffen.

**Level 2: HTTP-Verben** Systeme auf Level 2 verwenden nicht nur mehrere Ressourcen, sondern auch die vorhandenen HTTP-Verben, um einen Dienst anzubieten. Dabei werden die HTTP-Operationen entsprechend ihrer Semantik eingesetzt, sodass HTTP nicht mehr nur als Tunnel-Mechanismus benutzt wird. Zusätzlich werden auf diesem Level die HTTP-Statuscodes verwendet.

**Level 3: Hypermedia** Ein System auf der dritten und letzten Stufe nutzt Hypermedia as the Engine of Application State (HATEOAS), um einen Client durch Referenzen von Ressource zu Ressource zu führen. Ein großer Vorteil dabei ist, dass bei richtiger Umsetzung der Server seine Ressourcenstruktur ändern kann, ohne dass Clients entsprechend geändert werden müssen.

## 2.3 TOSCA

OASIS[1] TOSCA (**T**opology and **O**rchestration **S**pecification for **C**loud **A**pplications) bezeichnet eine Spezifikation für Cloud-Anwendungen, die eine Automatisierung des Deployments und der Verwaltung solcher Applikationen zum Ziel hat. Ferner sollen TOSCA-basierte Anwendungen möglichst portabel sein. Durch das Fehlen einer bei Cloud-Dienstleistern häufiger vorkommenden Herstellerabhängigkeit (Vendor-Lock-In) soll somit ein herstellerneutrales „Ökosystem“ geschaffen werden.

### 2.3.1 Topology

Im Sinne von TOSCA bezeichnet ein *Topology Template* diejenige Struktur, die nötig ist, um einen bestimmten Dienst anzubieten (service structure) [14]. Es besteht aus einer Menge von *Node Templates* und *Relationship Templates* (Abbildung 2.5).

[1]Organization for the Advancement of Structured Information Standards, ein gemeinnütziges Konsortium, das sich mit offenen Standards für Informationstechnologien beschäftigt. Siehe auch: `https://www.oasis-open.org/`

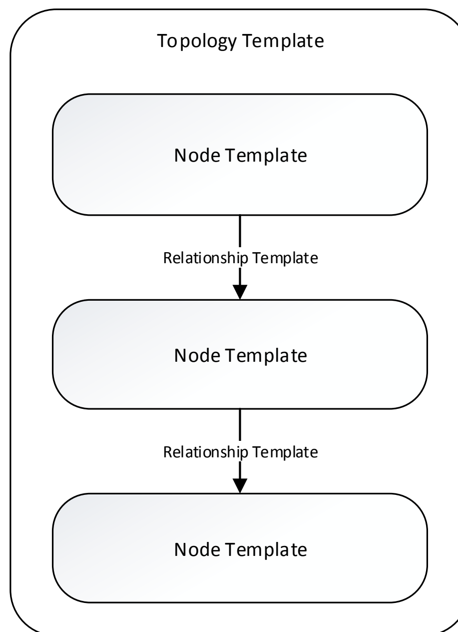


Abbildung 2.5: Ein TOSCA Topology Template und seine Komponenten

Ein Node Template bezeichnet die einzelnen (abstrakten) Komponenten eines Services. Ein solches Node Template ist von einem bestimmten Typ, dem sogenannten *Node Type*. Zu einem Node Type wiederum gehören bestimmte *Properties* und *Operationen*. Zwischen Node Templates können durch *Relationship Templates* Beziehungen beschrieben werden. Ähnlich wie bei den Nodes bezeichnet auch hier ein Relationship Template eine konkrete Ausprägung eines *Relationship Types*.

Abbildung 2.6 zeigt ein sehr einfach gehaltenes, nicht vollständiges Beispiel einer möglichen Modellierung einer Webshop-Anwendung. Hierbei gibt verschiedene Elemente:

- fünf Node Types mit fünf Node Templates, ihren Properties und deren Werte:

    1. KVM Virtual Server

        - IP address = 129.69.123.42

    2. Debian Operating System

        - version = Debian Linux 8.1

    3. Tomcat Webserver

        - username = admin

  - password = sd?lO!a23

4. MySQL Database Server
   - username = webshop
   - password = vcxo234!dl4
5. Webshop WAR-Archive

- fünf Relationship Templates und deren zwei Relationship Types „hosted-on“ und „accessed-by“

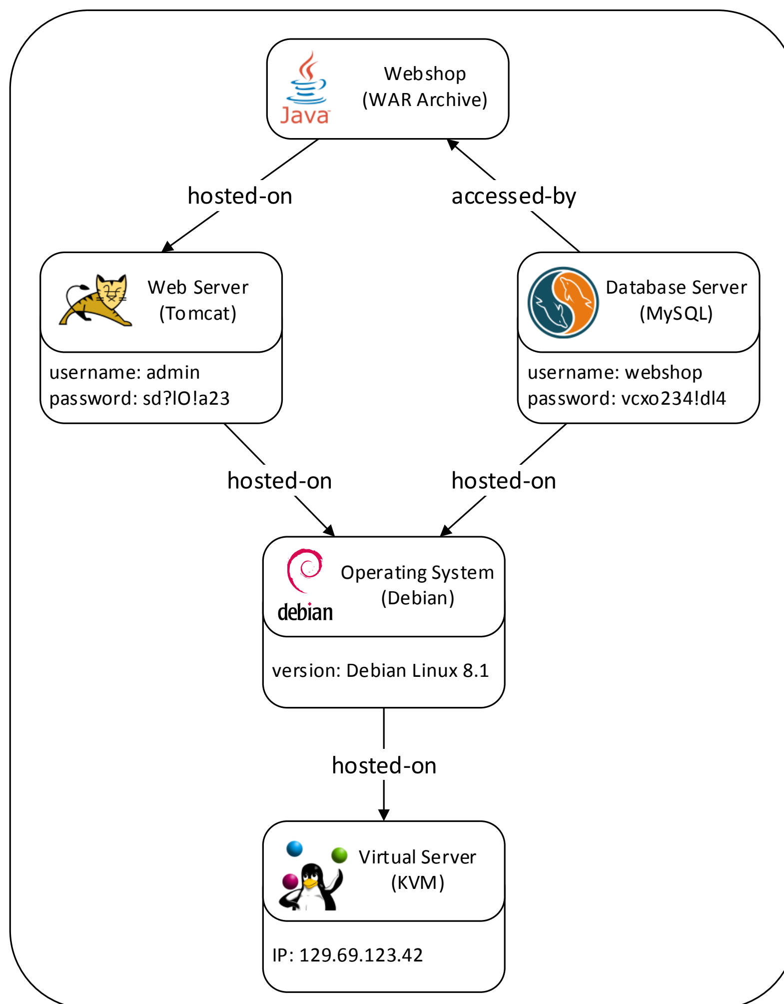


Abbildung 2.6: Beispiel einer TOSCA Topology für einen Webshop

Mögliche Operationen etwa für den Tomcat Webserver wären zum Beispiel „start“ oder „stop“, um den Zustand des Servers zu kontrollieren. Für das Betriebssystem hingegen könnte man sich eine generische „execute-command“-Operation vorstellen,

mit der ein bestimmter Befehl ausgeführt wird. Auch passend wäre ein „install-package“, um weitere Anwendungen über eine Softwareverwaltung nachzuinstallieren.

Die jeweilige Funktionalität eines Nodes wird durch sogenannte Artefakte bereitgestellt. Artefakte können etwa Scripte, ausführbare Dateien, Images, Konfigurationsdateien, oder Bilder sein. Die oben genannten Operationen werden durch die entsprechenden Artefakte implementiert.

### 2.3.2 Orchestration

Als Orchestration bezeichnet man den Vorgang des Ausbringens (Deployment) sowie das Verwalten (Management) einer Topology. Hierbei werden zwei Arten der Orchestration unterschieden:

**deklarativ:** Bei dieser Art wird beschrieben, *was* (bzw. welches Ziel) erreicht werden soll. Die letztliche Umsetzung der einzelnen Schritte bis dahin wird von der Ausführungsumgebung („runtime“) übernommen. Beispielsweise könnte ein Ziel lauten: *Stelle mir einen Tomcat Webserver zur Verfügung*. Die genauen Abläufe (VM aufsetzen, Betriebssystem installieren, Tomcat installieren) werden dabei ausgeklammert und dem jeweiligen Container überlassen. Dieser Ansatz gestaltet einerseits sehr einfach, Services bzw. Topologien zu modellieren und bietet daher eine geringe Einstiegshürde, hat aber andererseits den Nachteil der geringen Anpassbarkeit bzw. Flexibilität. Gewissermaßen kann man diese Art als „Top-Down-Ansatz“ bezeichnen.

**imperativ:** Beim imperativen Ansatz wird jeder einzelne Schritt explizit beschrieben. Es werden also im Gegensatz zum imperativen Vorgehen viele Details behandelt. Dieser „Bottom-Up-Ansatz“ erlaubt eine höhere Flexibilität auf Kosten der leichten Modellierbarkeit. Die imperative Orchestration wird durch einen *Mangement Plan* beschrieben, der mit BPEL[1] oder BPMN[2] formuliert werden kann.

### 2.3.3 CSAR

CSAR (**C**loud **S**ervice **Ar**chive) bezeichnet das TOSCA-Format für ein Paket, das alle für das Deployment eines Services benötigten Komponenten enthält. Dazu gehören unter anderem die Topology Templates, die dazugehörigen Types, die Management Plans und die Artefakte einer Anwendung (s. Abbildung 2.7).

[1] Business Process Execution Language
[2] Business Process Model and Notation

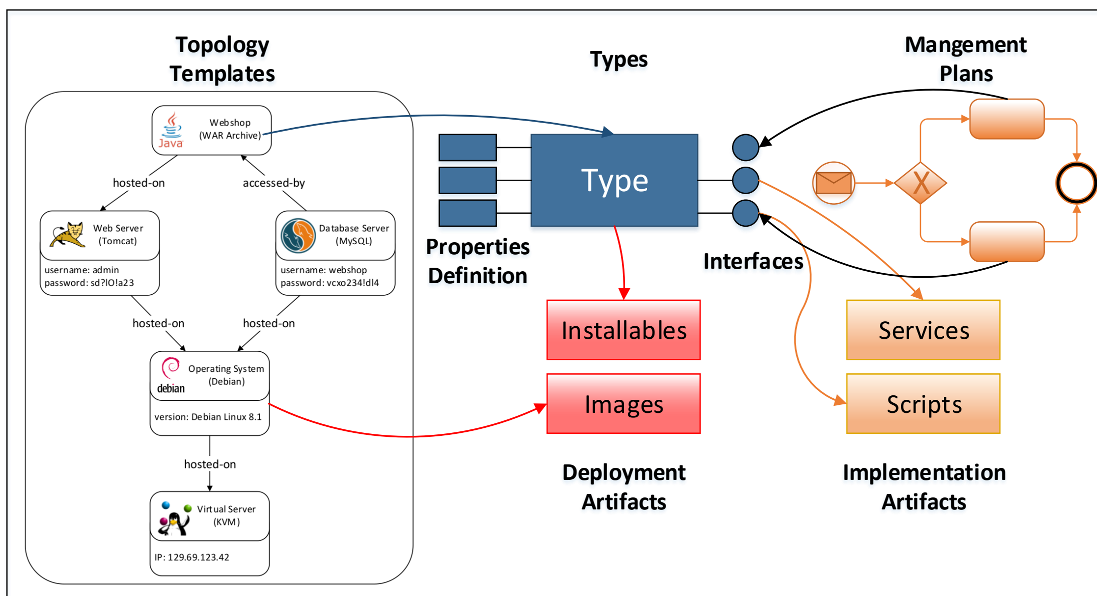


Abbildung 2.7: Inhalt eines Cloud Service Archivs (nach [17])

### 2.3.4 OpenTOSCA

OpenTOSCA bezeichnet ein an der Universität Stuttgart entwickeltes open-source „Ökosystem“, welches aus drei Komponenten besteht:

**Winery** ist ein webbasiertes Werkzeug, um TOSCA-Topologien und Management Pläne zu modellieren. Es ist als Standalone-Tool konzipiert und erlaubt den Import und Export als CSAR für eine TOSCA-Ausführungsumgebung.

**OpenTOSCA Container** bezeichnet eine TOSCA-runtime, also die Ausführungsumgebung (Container). Dies ist eine Middleware, welche ganze CSAR-Pakete verarbeiten, deren Pläne ausführen und Zustände verwalten kann.

**Vinothek** bezeichnet einen web-basierten „self-service“, der technische Details vor dem Endnutzer versteckt und eine einfache Oberfläche für das Provisionieren der Cloud-Anwendungen bietet.

Der später verwendete Begriff der Node Instance bezeichnet eine konkrete Instanz eines Nodes, gewissermaßen das „lebende“ Objekt innerhalb der TOSCA-Ausführungsumgebung.

# 3 Bisherige Lösung

Es existiert ein Vorprojekt, welches eine ähnliche Funktionalität wie das nun entwickelte Monitoring Backend bieten sollte. Im Folgenden wird dieses Vorprojekt kurz vorgestellt und auf ausgewählte Schwachstellen eingegangen.

## 3.1 Aufbau und Funktion

Das Vorprojekt hat den Namen „Monitor Server". Es wurde in Java entwickelt und wird als Web Application Archive (.war) beziehungsweise Java Servlet auf einem Tomcat-Server ausgebracht.
Das Ziel des Projektes war es, die bei der Provisionierung von Cloud-Anwendungen innerhalb der OpenTOSCA-Laufzeitumgebung auftretenden Prozesse nachvollziehbar zu machen. Dies geschieht durch das Schicken von Events an den Monitor Server. Dieser speichert die Events und macht sie für den späteren Zugriff abrufbar. Das Projekt besteht aus zwei Komponenten: Einem mit JavaScript umgesetzten Teil, welcher für die Darstellung der gespeicherten Events zuständig ist (Frontend), sowie dem Backend, welches die Speicherung der Events übernimmt. In der Abbildung 3.1 sieht man das Zusammenwirken der verschiedenen Komponenten.

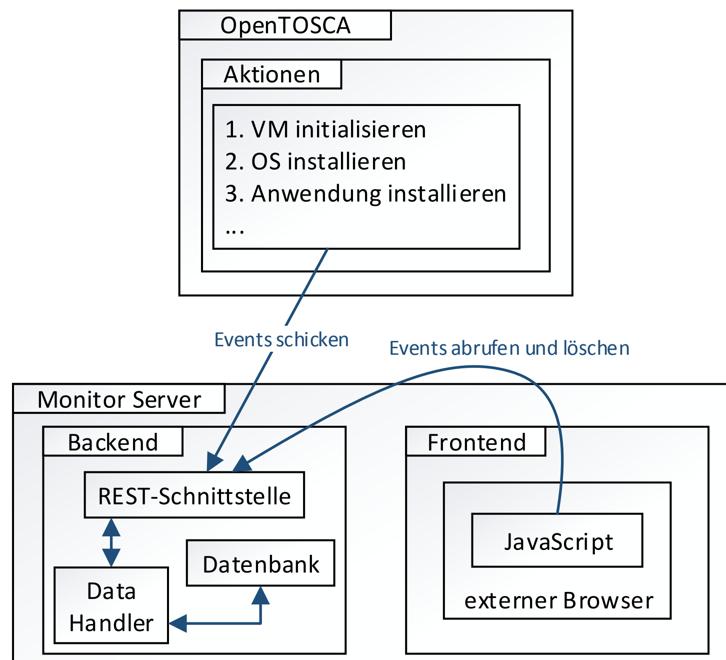


Abbildung 3.1: Zusammenhang zwischen den Komponenten von Monitor Server und OpenTOSCA (bisherige Lösung)

Dabei geschieht das Verarbeiten der Events wie folgt. OpenTOSCA generiert ein Event, welches Informationen über den aktuell laufenden Prozess beinhaltet. Dieses Event wird an die REST-Schnittstelle des Monitor Servers geschickt. Anschließend erfolgt eine Bearbeitung und das Abspeichern in eine Datenbank. Das Backend bietet zusätzlich noch die Möglichkeit, bereits gespeicherte Events über eine REST-Schnittstelle abzurufen. Die Darstellung erfolgt dann über das Frontend im Webbrowser.

## 3.2 Schwachstellen

Nach der Umsetzung stellte sich heraus, dass es einige Verbesserungsmöglichkeiten sowohl beim Design der REST-API, als auch bei der Implementierung gibt. Im Folgenden werden diese Schwachstellen mit prägnanten Beispielen näher erläutert.

**RESTfulness** Mit dem Begriff RESTfulness wird beschrieben, ob und inwieweit ein Dienst die REST-Prinzipien (siehe Abschnitt 2.1) befolgt. Eine Verletzung gibt es zum Beispiel bei der Umsetzung der DELETE-Operation: Die angebotene REST-Schnittstelle sieht für den Umgang mit Events genau eine Ressource vor: */collector*. Über die Operationen POST und GET können Events abgespeichert und abgerufen werden. Allerdings gibt es als dritte Operation noch DELETE. Mit dieser werden vorhandene Events und zu diesen angelegte NodeInstances gelöscht. Die Semantik der DELETE-Operation ist jedoch nicht das Löschen des Inhaltes einer Ressource (so wurde es hier umgesetzt), sondern das Entfernen einer ganzen Ressource aus dem „Ressourcenbaum". Somit ist die gewünschte Aktion mit DELETE falsch umgesetzt.

Diskutabel ist auch die Verortung der Ressourcen, die zur Verwaltung der als eigenständige Entitäten gespeicherten NodeInstances dienen. Diese werden unterhalb von */collector* eingeordnet. Damit sieht die Ressourcenstruktur wie folgt aus:

**/collector** Mit dieser Ressource werden Events verwaltet (Hinzufügen, Abrufen, Löschen).

**/collector/opentoscadata** Diese Ressource verwaltet Informationen über eine NodeInstance (Hinzufügen, Abrufen).

**/collector/opentoscaproperties** Über diese Ressource werden die OpenTOSCA-Properties einer NodeInstance gesondert zur Verfügung gestellt.

Klar erkennbar ist hierbei, dass die Verwaltung der NodeInstances unterhalb der Verwaltung der Events angesiedelt ist, obwohl eine NodeInstance kein eigenständiger Teil eines Events (im Sinne einer Subressource) ist.

Angemerkt sei jedoch, dass jedem Event die Identifikation einer NodeInstance zugeordnet ist. Diese Information dient der Zuordnung, von welcher NodeInstance das Event geschickt wurde.

**Erweiterbarkeit** Zwar gibt es im Code Ansätze, welche die Erweiterbarkeit begünstigen, jedoch sind diese nicht konsequent umgesetzt. So wurde zum Beispiel das Abspeichern von Events in eine Datenbank von der REST-Schnittstelle selbst getrennt, jedoch gibt es keine Möglichkeit, die Art der Speicherung (etwa flüchtig, als lokale Datei, oder in eine (entfernte) Datenbank) komfortabel zu ändern. Ebenfalls schwierig gestaltet sich die Erweiterung der Logging-Funktionalität: Es können ausschließlich OpenTOSCA-Events verarbeitet werden, ein anderer Typ von Events ist nicht vorgesehen und kann nicht einfach hinzugefügt werden. Sehr ähnlich verhält es sich mit mehreren „collectors", also für sich abgeschlossenen Teilen, die jedoch die gleiche Funktionalität bieten. So ist es nicht ohne größeren Aufwand möglich, zum Beispiel einen zweiten OpenTOSCA-Logger hinzuzufügen.

**Funktionalität** Beim Abfragen der Events kann die Ausgabe gefiltert werden. Dies geschieht über einen optionalen Query-Parameter im URL. Jedoch kann hier ausschließlich die ID der zurückgegebenen Events eingeschränkt werden. Zurückgegeben werden dann nur solche Events, deren ID größer als die angegebene ist („eventAfterID").

**Modularität, Kopplung, Zusammenhalt** Die Modularität spielt bei der Betrachtung aus softwaretechnischer Sicht eine größere Rolle. Sie hat einen nicht unerheblichen Einfluss auf die Anpassbarkeit des Codes und ist eng verbunden mit dem Prinzip der *Separation of Concerns*. Hierbei wird versucht, verschiedene Elemente der Aufgabe möglichst auf verschiedene Elemente der Lösung abzubilden. Dies führt unter anderem dazu, dass Änderungen nur die Komponente betreffen, die geändert wurde. Ein oft eingesetztes Mittel hierzu sind Schnittstellen, welche die Möglichkeiten des Zugriffes regeln, Implementierungsdetails jedoch wegkapseln. Generell ist die Frage nach der Modularität eng mit der Erweiterbarkeit verbunden. Die Kopplung zweier Softwarekomponenten bezeichnet die Stärke der Verbindung und damit der Abhängigkeiten untereinander. Der Zusammenhalt einer beschreibt, wie groß die Zusammengehörigkeit einzelner Bestandteile in einer Softwarekomponente ist. Wünschenswert ist somit eine geringe Kopplung und ein starker Zusammenhalt. In Abbildung 3.2 ist ein solches System mit geringer Kopplung (wenige Verbindungen zwischen den Gruppen) und hohem Zusammenhalt (viele Verbindungen innerhalb einer Gruppe) dargestellt.

In Bezug auf das hier diskutierte System gibt es im Speziellen zwei Auffälligkeiten: Operationen, im Besonderen solche auf einer Ressource, werden letztlich zum Großteil nur durchgereicht. Dies erhöht die Kopplung und den Aufwand, um Anpassungen durchzuführen. Ferner ist die Darstellung der Ereignisse (Frontend) im selben Projekt wie deren Speicherung (Backend) zusammengeführt.

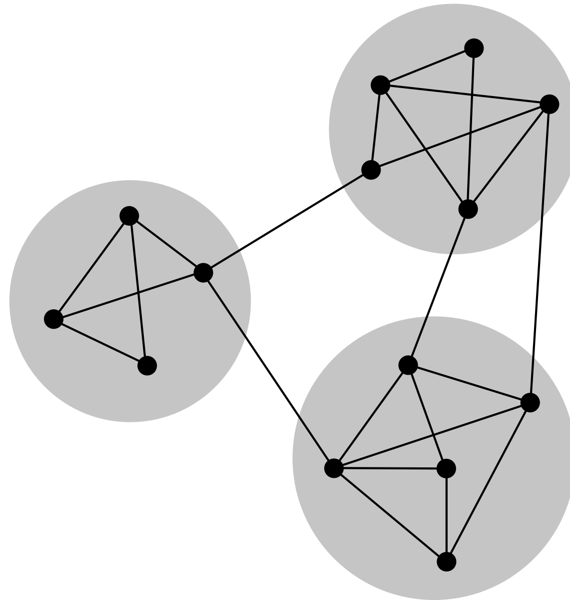

Abbildung 3.2: System mit geringer Kopplung und hohem Zusammenhalt (Grafik aus [18])

# 4 Entwurf des Monitoring Backends

In diesem Abschnitt wird die Entwurfsphase des entwickelten Monitoring Backends näher beleuchtet. Zunächst wird dabei auf den Entwurf der REST-Schnittstelle eingegangen, die genutzten Methode und das dazugehörige Modellierungswerkzeug vorgestellt und anschließend der Entwurf des Grundsystems beschrieben. Des Weiteren wird darauf eingegangen, wie das Grundsystem erweitert werden kann. Dies wird abschließend am Beispiel einer OpenTOSCA-Erweiterung gezeigt.

Allgemein sei angemerkt, dass es ein separates Parallelprojekt gibt, in welchem die Darstellung der Events für den Anwender, also das Frontend, entwickelt wird. Obwohl das Frontend im Rahmen dieser Arbeit nur als ein vom Backend abgekapseltes Modul auftritt, hatte es einen nicht unerheblichen Einfluss auf die Gestaltung der REST-Schnittstelle. Als Client stellt es letztlich die Anforderungen an die REST-API und bestimmt damit zu einem Teil, die diese aussieht. Ein Beispiel für diese Verknüpfung der beiden Projekte ist die in Abschnitt 4.2 näher beschriebene Liste der Eventquellen.

Die grundsätzliche Idee des Monitoring Backends bleibt dieselbe wie beim Vorgängerprojekt. Von einer Quelle, etwa einer OpenTOSCA-NodeInstance, wird ein Event an das Backend geschickt. Dieses Event wird anschließend persistent gespeichert. Schließlich werden die empfangenen Events für die spätere Abfrage und Darstellung verfügbar gemacht.

## 4.1 Entwurf der REST-API

Bei der Entwicklung des Monitoring Backends spielte die Erweiterbarkeit von Anfang an eine große Rolle. Dies spiegelt sich auch in der REST-API wider, welche das Bindeglied zwischen dem Ursprung der Events, eines entsprechenden Frontends zur Darstellung und der eigentlichen Speicherung darstellt.

### 4.1.1 Anforderungen

Die REST-Schnittstelle des Backends ist die wesentliche und einzige Verbindung nach außen hin. Sie soll den Zugang zu folgenden Funktionen regeln:

**Events empfangen:** Mehrere einzelne Informationen wie etwa die Kennung der Quelle oder das Datum stellen im Verbund ein Event dar. Solche Events sollen von einer Quelle (*Source*) an die REST-API geschickt werden können.

**Events anbieten** Die zuvor gespeicherten Events sollen über die REST-API zugänglich gemacht werden, sodass Clients, wie etwa ein Frontend, diese abfragen und weiterverarbeiten können.

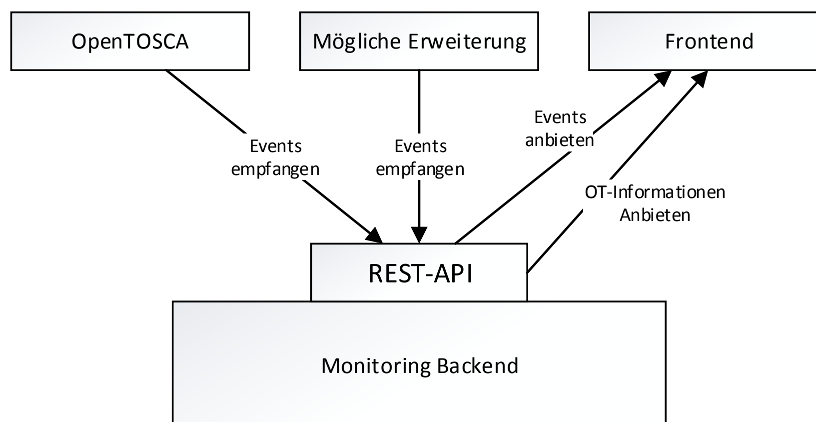


Abbildung 4.1: Aufgaben der REST-API des Monitoring Backends

**OpenTOSCA Informationen anbieten:** Eventquellen sind bei OpenTOSCA sogenannte NodeInstances. Eine NodeInstance hat verschiedene Attribute (etwa die CSAR ID oder einen Status), welche ebenfalls verfügbar gemacht werden sollen. Dies ermöglicht Clients, sich auf den Inhalt der Events und NodeInstance-Attribute zu konzentrieren und Implementierungsdetails der ihrer an das Backend zu übertragen.

Die Schnittstelle muss im Besonderen den folgenden zwei Anforderungen gerecht werden, um eine möglichst hohe Modularität des Gesamtsystems zu ermöglichen.

**RESTfulness:** Die REST-API soll RESTful sein, das heißt, sie soll die in Abschnitt 2.1 vorgestellten Beschränkungen (Constraints) einhalten. Dazu zählen vor allem *Uniform Interface* und *Stateless Server*. Die Idee dahinter ist eine auf lange Sicht leichtere Skalierbarkeit, da Systeme, die sich an diese Constraints halten, wachsen können, ohne diese Eigenschaften zu verlieren. Diese Anforderung geht Hand in Hand mit der Forderung nach Erweiterbarkeit.

**Erweiterbarkeit:** Eine weitere Anforderung ist die Möglichkeit, das vorhandene System mit möglichst geringem Aufwand erweitern zu können. Dies ermöglicht das rasche und unkomplizierte Anpassen an neue Begebenheiten.

## 4.2 Umsetzung des API-Entwurfs

Um die Anforderungen zu erfüllen, werden unter anderem *Storages* eingeführt. Ein Storage ist ein Speicher für einen bestimmten Typ von Events. Zu einem Storage gehören weitere Attribute, die ihn charakterisieren. Ferner ist ein Teil der REST-API storagespezifisch.

Dieses Konzept ermöglicht sowohl eine Trennung der Log-Funktionalität logisch abgeschlossener Prozesse, als auch das unkomplizierte Erweitern um einen weiteren Event-Typ (Welche Mechanismen dieses Erweitern erlauben, wird in Abschnitt 4.3 ausgeführt). Eine mögliche Ausprägung eines Storages ist zum Beispiel ein OpenTOSCA-Storage, welcher entsprechend OpenTOSCA-Events zu speichern und anzubieten vermag. Es kann mehrere Storages des gleichen Typs geben.

In Abbildung 4.2 ist der Aufbau der REST-API dargestellt. Hierbei werden Erweiterungen wie die später hinzugefügte OpenTOSCA-Erweiterung nicht betrachtet. Gezeigt wird die API mit dem *Basic-Storage*. Dies ist der Storage-Typ, von welchem die Erweiterungen (andere Storage-Typen) abgeleitet werden. Diese Hierarchie ermöglicht einen einheitlichen Zugriffsmechanismus auf Storages unterschiedlichen Typs. Näheres hierzu entnimmt man Abschnitt 4.4.
Der Entwurf wurde mit Hilfe eines an der Universität Stuttgart entwickelten Werkzeugs [5] ausgearbeitet. Dieses ermöglicht das Modellieren einer REST-Schnittstelle sowie die Generierung des eigentlichen Programmcodes aus dem erstellten Modell. Dadurch können Fehler bei der späteren Implementierung vermieden werden, ohne jedoch manuelle Anpassungen der Programmlogik zu verhindern. Ein großer Vorteil dieses Ansatzes liegt darin, dass durch die automatisierte Erstellung (eines Teils) der Implementierung die Konformität mit den REST-Constraints weitgehend gegeben ist und auch bei Änderungen bleibt.

Die modellierten Ressourcen haben dabei folgende Aufgabe:

**Home:** Über diese Ressource ist eine Abfrage aller vorhandenen Event-Storages möglich. Zu den jeweiligen Storages werden Informationen wie der Storage-Typ oder ein Link zur Verfügung gestellt. Dies ermöglicht einem Client, sich einen Überblick über die Storages zu verschaffen.

**Basic Event Storage:** Diese Ressource stellt den eigentlichen Basic-Storage dar. Zur Verfügung gestellt werden hierüber hauptsächlich Informationen, wo sich die weiteren Funktionen des Storages befinden. Dies geschieht durch Links auf die entsprechenden Ressourcen, z.B. `link:event=/basic/event/{eventID}`, und folgt damit dem HATEOAS-Constraint.

**Event List:** Die *eventlist*-Listenressource stellt das Kernstück der API dar. Mit ihr kann man Events speichern und eine Liste der bereits empfangenen Events abrufen. Entsprechend den in [4] diskutierten Problemen großer Listenressourcen (*collection resources*) wird es ermöglicht, über Query-Parameter die zurückgegebenen Events etwa nach ihrer Quelle zu filtern.

**Event:** Über diese Ressource lassen sich weitere Informationen zu einem durch die „eventID“ eindeutig bestimmten Event abfragen. Auch können einzelne Events gelöscht werden.

**Event Dropper:** Diese Ressource dient hauptsächlich dem Komfort beim Testen,

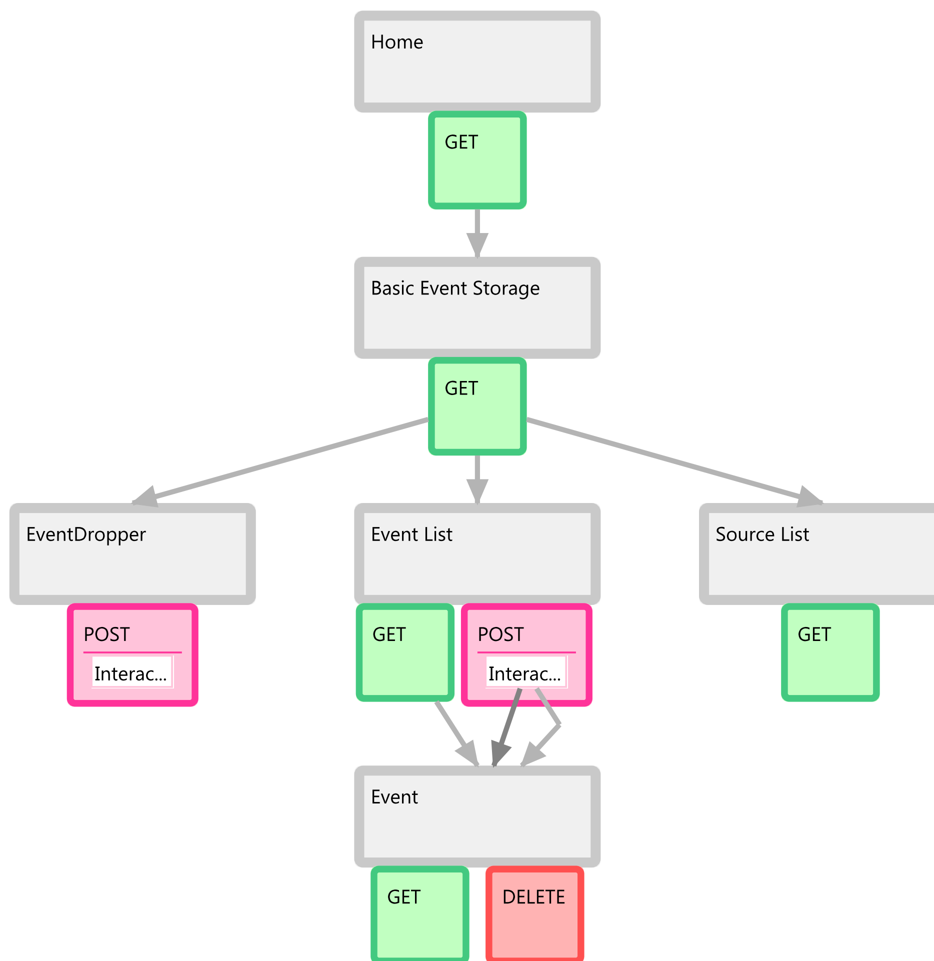


Abbildung 4.2: REST-Schnittstelle für den *Basic-Storage*

Anpassen und Integrieren des Monitoring Backends. Mit ihr ist es möglich, mehrere, zuvor gespeicherte Events zu löschen. Diese Funktionalität kann nicht von der *eventlist*-Ressource übernommen werden, da die Semantik DELETE-Operation nicht zur gewünschten Löschung der Events passt. Eine Aktualisierung von Ressourcen mittels PUT auf *eventlist* wäre eine Alternative, von der jedoch abgesehen wurde, da weniger die Liste der Events modifiziert werden soll, als vielmehr die Events selbst (sie werden gelöscht). Dies führte zur Entscheidung, die *eventdropper*-Ressource einzuführen, auf der mittels POST operiert werden kann.

**Source List:** Zu einem Event gehört immer auch eine Quelle, die das jeweilige Event geschickt hat. Für die Darstellung der Events durch einen Client (Frontend) kann es sehr hilfreich sein, alle vorhandenen Quellen zu kennen. Um möglichst viel Funktionalität an das Backend zu übertragen, wurde die *sourcelist*-Ressource eingeführt, welche alle bisher aufgetretenen Eventquellen zur Verfügung stellt. Ferner erlaubt dieser Ansatz es der GUI, sehr leicht herauszufinden, nach welchen Sources man filtern kann. Ein weiterer Vorteil

ergibt sich bei einer Darstellung der Events, bei welcher diese nach nach Eventquelle separiert werden: Das Fontend muss so nicht selbst alle vorhandenen Events abfragen, durchgehen und eine Liste der Sources erstellen, sondern kann diese Liste einfach beim Backend erfragen.

### 4.2.1 Use Cases

In Abbildung 4.3 sieht man die verschiendenen Use Cases, die sich aus der Sicht eines Clients mit dem Monitoring Backend ergeben können. Die Interaktionsmöglichkeiten einer Eventquelle sind vergleichsweise beschränkt. Durch die Abfrage der vorhandenen Storages und genauerer Informationen über diese kann sie ein Event an den Server schicken bzw. es abspeichern. Ein Frontend hingegen, das die Events darstellt, hat wesentlich mehr Möglichkeiten, die REST-API zu nutzen. So können Events auf unterschiedlichen Arten abgefragt werden: Man kann die zurückgegebene Liste nach bestimmten Kriterien filtern oder auch sortiert erhalten. Ebenfalls kann man die bereits bekannten Eventquellen oder auch nur Informationen zu einem einzelnen, bestimmten Event erfragen. Ferner können ein oder mehrere Events gelöscht und zu den Storages spezifische Informationen abgefragt werden. Wie eine Quelle auch, kann ein Frontend natürlich die vorhandenen Eventstorages in Erfahrung bringen.

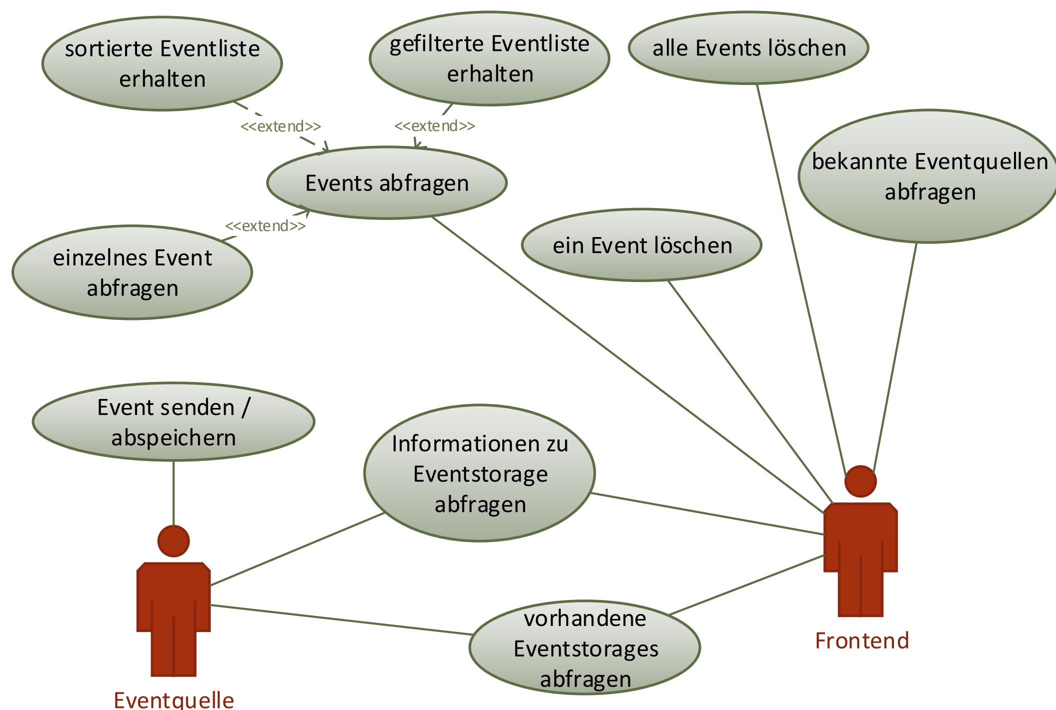


Abbildung 4.3: Use Cases aus Sicht von Frontend und Eventquelle

### 4.2.2 Mechanismen bei der API-Interaktion

Es gibt mehrere Möglichkeiten, um über eine HTTP API Daten zu übertragen. Zwei Arten sollen im Folgenden näher betrachtet werden:

**Daten im Message-Body:** Hierbei befinden sich die zu übertragenen Daten im Anfragekörper (Request Body). Das Format, wie die Daten dargestellt werden, ist nicht festgelegt und dann beliebig definiert werden. Bei Events könnte eine Beispielanfrage wie folgt aussehen. Dabei wird implizit davon ausgegangen, dass die erste Zeile das Datum eines Events enthält, die zweite Zeile die Quelle und die dritte Zeile die eigentliche Nachricht des Events.

**Listing 4** Event-Daten im Message-Body

```
1 POST /eventstorages/basic/eventlist HTTP/1.1
2 Host: www.opentosca.org
3
4 2016-02-15
5 operating system
6 booted
```

**Daten über Query-Parameter:** Hierbei wird die Nutzlast als Anfrage-Parameter codiert der ersten Zeile des Message-Bodys hinzugefügt. Diese Codierung entspricht der Angabe von Schlüssel-Wert-Paaren. Eine implizite Semantik, etwa der Zeilennummer, kann es hierbei nicht geben. Der obige HTTP-Request sähe zum Beispiel so aus:

**Listing 5** Event-Daten als Query-Parameter

```
1 POST /eventstorages/basic/eventlist HTTP/1.1
2 Host: www.opentosca.org
3
4 date=2016-02-15&source=operating%20system&message=booted
```

Angemerkt sei jedoch, dass es durchaus möglich ist, zusätzlich zu den Query-Parametern weitere Daten in den Body zu schreiben.

Beim Monitoring Backend fiel die Entscheidung bewusst auf die Datenübergabe als Query-Parameter. Einer der Hauptgründe ist die Rückwärtskompatibilität mit Systemen, die bereits in Verwendung sind. Das Vorgängerprojekt setzte auf Query-Parameter, weshalb ein Austausch des Backends ohne Anpassen der Quelle/Clients nur möglich ist, wenn ebenfalls Anfrage-Parameter genutzt werden. Ein weiterer Grund ist die leichtere Nutzung des Monitoring Backends von der Kommandozeile aus oder durch Scripte, da die entsprechenden Daten nur dem Anfrage-URL hinzugefügt werden müssen. Das Nutzen eines eigenen Formates (oder auch von JSON) ist dagegen etwas aufwändiger und macht das Programmieren eines Clients nur unnötig komplexer.

Die REST-API bietet nur ein Format (Repräsentation) für die Inhalte an. Dies ist die JavaScript Object Notation (JSON), da dieses Format für das als Parallelprojekt entwickelte Frontend am passendsten ist. In diesem wird ausgiebig Gebrauch von JavaScript gemacht, weshalb die Nutzung von JSON naheliegt.

### 4.2.3 URL-Struktur der REST-API

In der Abbildung 4.2 wurden die Ressourcen durch Namen identifiziert. Um letztlich auf sie zugreifen zu können, bedarf es einer Abbildung der Namen auf die jeweilige URL, unter der die Ressource erreichbar ist. Diese URL-Struktur wird im Folgenden kurz aufgezeigt:

**Home:** `/eventstorages`
Home bezeichnet den Einstiegspunkt des Monitoring Backends. Hier kann der Client die vorhandenen Event-Storages erfragen und erhält einen Verweis auf den jeweiligen Storage (HATEOAS).

**Basic Event Storage:** `/eventstorages/basic`
*basic* bezeichnet den Event-Storage. Er wird unterhalb der *eventstorages* verortet.

**Event List:** `/eventstorages/basic/eventlist`
Die Event List gehört zum Basic-Storage und ist entsprechend unterhalb davon angesiedelt.

**Event:** `/eventstorages/basic/event/{eventID}`
Über den Pfad-Parameter *eventID* kann auf ein bestimmtes Event zugegriffen werden.

**Event Dropper:** `/eventstorages/basic/eventdropper`
Diese Ressource wird zum Löschen aller Events genutzt. Da sie sich das Löschen auf den gesamten Storage bezieht, ist sie direkt unterhalb von ihm zu finden.

**Source List:** `/eventstorages/basic/sourcelist`
Bei später hinzugefügten Erweiterungen soll die jeweilige Liste vorhandener Quellen nur genau solche Quellen einschließen, die bei Events des jeweiligen Storage-Typs aufgetreten sind. Dadurch liegt die Positionierung unterhalb eines Storages nahe.

## 4.3 Entwurf des Grundsystems

Als Grundsystem wird im Folgenden derjenige Teil hinter der REST-API bezeichnet, der über die Schnittstelle zur Verfügung gestellte Funktionen umsetzt.

Beim Entwurf dieses Grundsystems spielen selbstverständlich dieselben Anforderungen wie bei der REST-API eine große Rolle. Es wurde entsprechend mit dem Ziel der leichten Erweiterbarkeit entworfen und umgesetzt. Um dies zu erreichen, wurde eine Trennung der einzelnen Komponenten eingeführt. Im Folgenden wird diese Aufteilung näher erläutert und an wenigen konkreten Beispielen ausgeführt, wie die jeweiligen Teile miteinander zusammenarbeiten.

### 4.3.1 Architektur

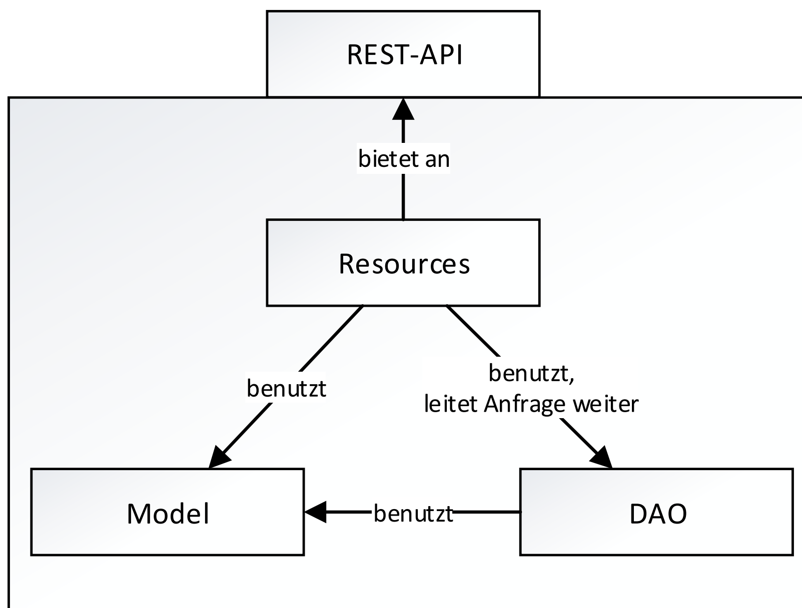


Abbildung 4.4: Architektur des Grundsystems (Basic-Storage)

Angelehnt an das Vorgängerprojekt gibt es eine Komponente „Resources'", welche die eigentliche REST-API umsetzt und darstellt. Ihre Aufgabe ist es, die verschiedenen Ressourcen unter einer logisch zusammenhängenden Einheit zu bündeln, sowie die Operationen auf den jeweiligen Ressourcen zu definieren. Umgesetzt werden diese Operationen entsprechend der geforderten Modularität durch ein Weiterleiten an den Teil des Systems, der sich um die (persistente) Speicherung kümmert.
Dieses *Data Access Object* (DAO) ermöglicht eine Abstrahierung des Zugriffes auf das zugrunde liegende Speichermedium. So ist es möglich, den jeweiligen Speicher (etwa eine Datenbank oder Dateisystem) auszutauschen, ohne dass der aufrufende Code modifiziert werden muss. Dieses Entwurfsmuster ist auch bei Tests sehr hilfreich, da so ein isoliertes Testen einzelner Komponenten möglich ist, und nicht alle Abhängigkeiten zur Verfügung stehen müssen.
Die dritte große Komponente ist das verwendete Datenmodell, zum Beispiel das

Modell der Events, die gespeichert werden. Dieses wird sowohl von den Ressourcen, als auch vom DAO benutzt.

### 4.3.2 Funktionsweise

In diesem Abschnitt soll das Zusammenwirken der Komponenten erläutert werden. Hierzu werden in Abbildung 4.5 die Schritte gezeigt, die clientseitig nötig sind, um ein Event an einen bestimmten Storage zu schicken und anschließend die Events wieder abzurufen. Hierbei liegt der Fokus auf der Kommunikation zwischen Client und der REST-API. Anschließend wird in Abbildung 4.6 darauf eingegangen, welche Teile des Grundsystems bei der Abfrage der Events welche Aufgabe erfüllen.

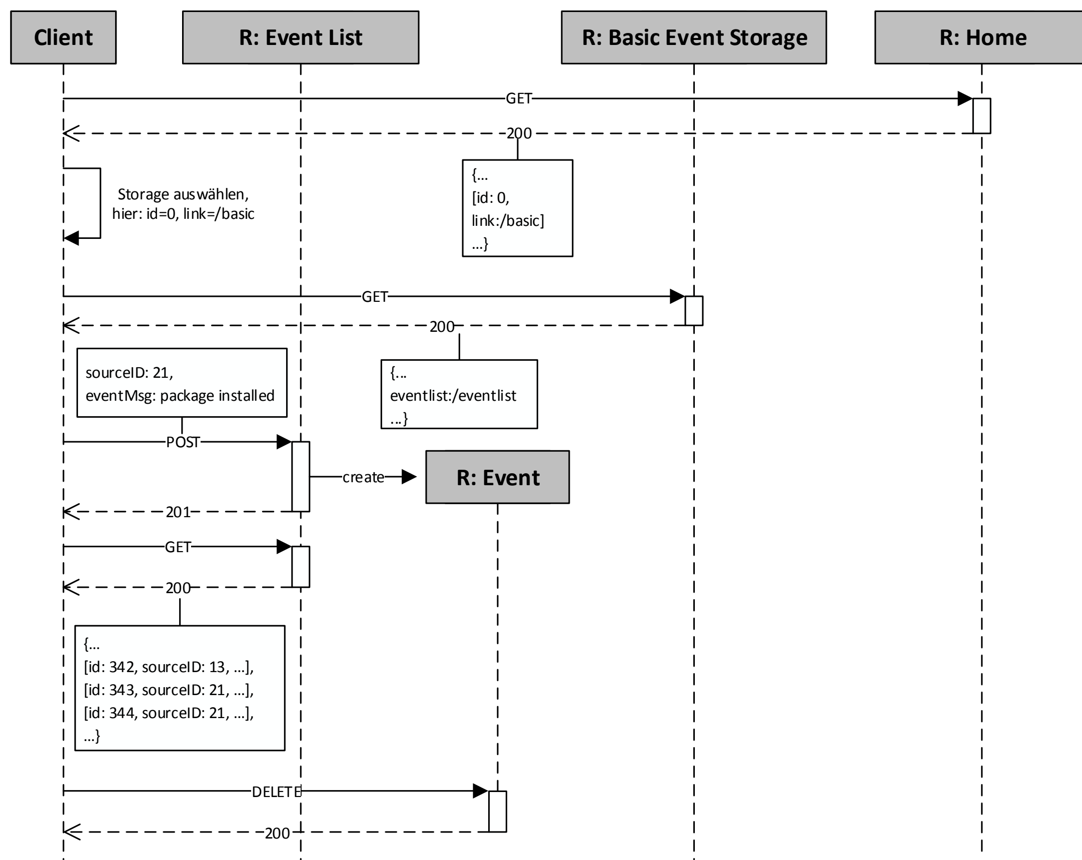


Abbildung 4.5: Speichern und Löschen eines Events über die REST-API

Ein Client möchte ein Event speichern, eine Liste der vorhandenen Events abfragen und abschließend das gespeicherte Event wieder löschen. Hierzu fragt er am Einstiegspunkt („home document“ [4]) nach, welche Storages überhaupt existieren. Zurückgegeben wird eine Liste der vorhandenen Storages, sowie weiteren

Informationen zu ihnen. Das in Kapitel 2.1.1 näher erklärte HATEOAS-Prinzip ermöglicht hierbei das Navigieren von einer Ressource zur nächsten, indem zurückgegebenen Links gefolgt wird. Entsprechend wird nach der Auswahl eines bestimmten Storages die entsprechende Ressource (hier: Basic Event Storage mit dem URL `/eventstorages/basic`) angefragt. Dadurch kommt man an einen weiteren Link, der zur Event-Listenressource dieses Storages führt. An diese Listenressource kann nun ein Event geschickt werden. In der Abbildung geschieht dies mittels POST. Diese Operation führt zum Anlegen einer neuen Ressource, die das Event selbst darstellt. Anschließend wird über ein GET auf die Listenressource eine Liste bereits gespeicherter Events zurückgegeben. Mit einem DELETE auf die zuvor angelegte Ressource, die das Event darstellt, wird diese gelöscht. Das dahinterstehende Event taucht dann auch nicht mehr in der Listenressource auf.
Die Prozedur des Erfragens der jeweiligen Ressourcen kann natürlich durch entsprechende Zwischenspeicherung (Caching) bei zeitlich nahen Operationen vermieden werden.

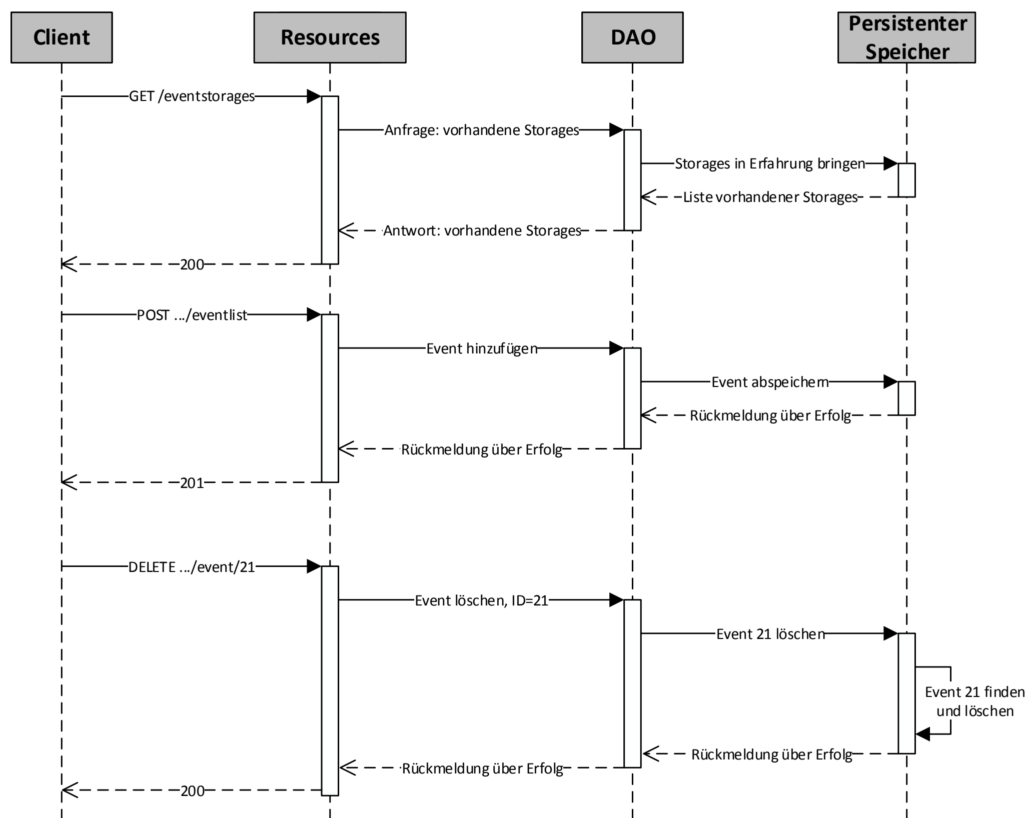


Abbildung 4.6: Kapselung der Systemkomponenten

In Abbildung 4.6 ist zu sehen, wie sich die Anfragen an die REST-API von oben aus Sicht des Grundsystems darstellen. Die einzelnen Ressourcen, welche die REST-Schnittstelle ausmachen, sind in „Resources“ zusammengefasst. Das DAO

abstrahiert die Verwaltung der Events und Storages vom jeweiligen persistenten Speicher.
Wird nun eine Anfrage gestellt, reichen die Ressourcen diese Anfrage weiter an das DAO. Die erfragten Daten werden dann von der jeweiligen DAO-Implementierung vom persistenten Speicher gelesen und zurückgegeben. Abschließend wird das Ergebnis an den Client zurückgeschickt. Nicht nur lesende Operationen, sondern auch schreibende (neu Erstellen und Löschen) werden über diesen Mechanismus umgesetzt.
Sehr deutlich wird hierbei, wie einzelne Aufgaben auch auf einzelne Komponenten aufgeteilt werden („Separation of Concerns“): Die REST-API mit ihren Ressourcen wird vollständig von der *Resources*-Komponente übernommen, während die Verwaltung der Daten dem DAO (beziehungsweise dessen Implementierung) obliegt. Die Dieses Konzept ermöglicht die geforderte hohe Anpassbarkeit. Zum Beispiel können so schnell weitere Ressourcen hinzugefügt oder vorhandene umbenannt werden, ohne die dahinterliegende Anbindung ändern zu müssen. Auch ist ein Austausch des Datenspeichers ohne Weiteres möglich, solange sich die Implementierung des neuen Speicherzugangs an die DAO-Schnittstelle hält.

### 4.3.3 Ableitung / Vererbung

Ein Kernelement, um die Erweiterbarkeit des Monitoring Backends umzusetzen, ist das Konzept einer Vererbungshierarchie. Eine Vererbung im softwaretechnischen Sinne bezeichnet die Erweiterung einer bestehenden Entität (etwa einer Klasse) um weitere Eigenschaften oder Funktionen. Der Vorgang der Vererbung wird auch als Ableitung bezeichnet. Die Entität, die vererbt, ist die Eltern-Entität, jene die erbt, die Kind-Entität.
Das Prinzip der Vererbung wird zum Beispiel bei den Storages eingesetzt. Dazu wird ein Basis-Storage definiert (der Basic-Storage), welcher bestimmte Grundeigenschaften besitzt, die allen Storages gemein sind. Betrachtet man einen Storage als Klasse einer (objektorientierten) Programmiersprache, so entspricht eine solche Grundeigenschaft einem Feld (Klassenattribut), das einen bestimmten Wert annehmen kann. Dies kann zum Beispiel der Link/URL zur Ressource *Event List* sein, die zu diesem Storage gehört. Wurde ein solcher Basis-Storage definiert, können Erweiterungen von diesem abgeleitet werden. Dasselbe Prinzip kommt bei den Events zum Einsatz. Es gibt ein Basis-Event, von welchem die Events der Erweiterungen erben. Exakt so wird bei der Implementierung des Grundsystems vorgegangen.
Durch die Unterscheidung von Eltern- und Kind-Klassen wird es möglich, Ressourcen wie die *Source List* unkompliziert umzusetzen. Bei dieser Ressource sollen nur die Eventquellen zurückgegeben werden, die Events vom Event-Typ eines bestimmten Storages geschickt haben. Da mit der in Abschnitt 5.1.6 vorgestellten Java Persistence API das Operieren auf Objektebene möglich ist, werden Abfragen etwa an eine Datenbank sehr vereinfacht.
Ferner erlaubt diese Herangehensweise die Programmierung eines generischen

Frontends, das keine Kenntnis über später hinzukommende, spezielle Event-Typen haben muss. Durch die Vererbung wäre ein solcher Client dennoch in der Lage, die Events (bzw. den Teil der Events, der allen gemeinsam ist) zu verarbeiten und korrekt anzuzeigen.
Die Idee der Vererbung findet sich nicht nur beim Grundsystem und dessen Umsetzung, sondern auch schon bei der REST-API. Die Ressource *Event List* des Base Storages soll alle Events (egal welchen Subtyps) verwalten. Event Lists anderer Storages hingegen dürfen nur innerhalb ihrer Domäne Events verwalten. Insbesondere sollen durch Operationen auf ihnen keine Events anderer Storages entfernen werden. Wie eine solche Vererbung durchgeführt wird, steht im folgenden Kapitel 4.4.

## 4.4 Entwurf einer Erweiterung: OpenTOSCA

Das gesamte Monitoring Backend wurde mit dem Gedanken der Erweiterbarkeit entwickelt. Im Besonderen soll es sehr leicht möglich sein, weitere Storages dem Gesamtsystem hinzuzufügen, ohne dabei die bisher existierenden ändern zu müssen. Ferner sollen solche Storages bereits vorhandene Funktionen übernehmen können, dabei jedoch eigene Änderungen oder auch Erweiterungen der Funktionalität erlauben. Im Folgenden wird der Entwurf einer solchen Erweiterung beschrieben. Es wird eine OpenTOSCA-Komponente entworfen, die die vom Vorprojekt teilweise umgesetzte Funktionalität bietet.

### 4.4.1 REST-API

Die REST-Schnittstelle entspricht im Groben der API des Basic-Storages. Zusätzlich zu den in Abschnitt 4.2 bereits beschriebenen Komponenten, welche dieselbe Funktionalität erfüllen, gibt es jedoch einige OpenTOSCA-spezifische Änderungen. An diesen ist gut erkennbar, wie sowohl Änderungen als auch das Hinzufügen neuer Ressourcen leicht möglich ist.

Einem OpenTOSCA-Event ist zwar ebenfalls eine Quelle zugeordnet, im Gegensatz zu einem Basic-Event heißt diese im OpenTOSCA-Kontext jedoch nicht „Source“, sondern *Node Instance*. Dies liegt in dem Wunsch begründet, dass die API den Konventionen der jeweiligen Domäne (hier: OpenTOSCA) folgt. Nutzern aus dem betreffenden Bereich sind Begriffe dieser Domäne bekannt, was das Verständnis der Schnittstelle mit entsprechenden Bezeichnern erleichtert. Entsprechend wird die Ressource, die zur Abfrage der bisherigen Eventquellen gedacht ist, umbenannt in */eventstorages/opentosca/nodeinstancelist*.

Ein HTTP-Request, der ein Event dem OpenTOSCA-Storage hinzufügt, sieht mit diesen Änderungen etwa wie folgt aus:

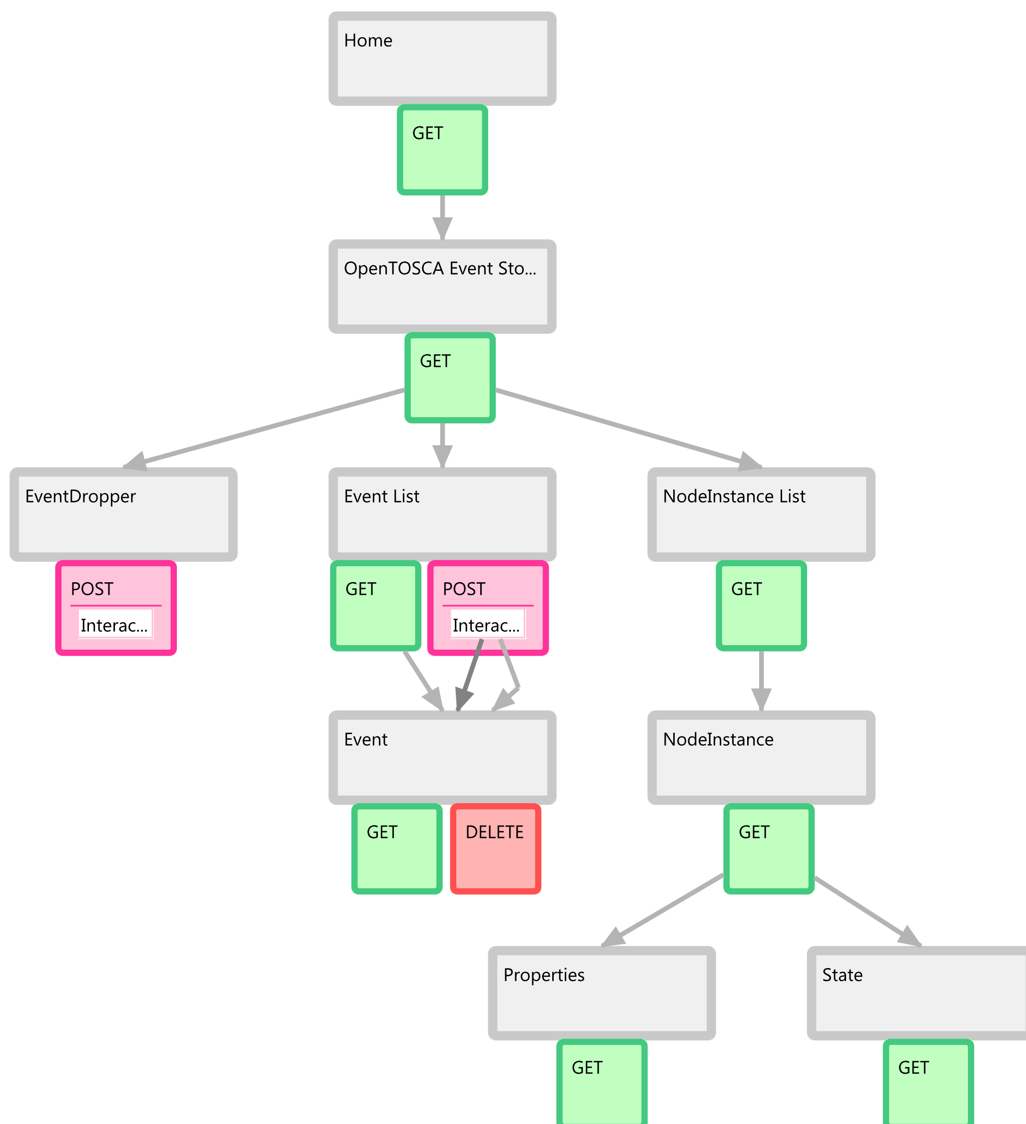


Abbildung 4.7: REST-Schnittstelle für die OpenTOSCA-Erweiterung

**Listing 6** HTTP-Request mit OpenTOSCA-Event als Nutzlast

```
POST /eventstorages/opentosca/eventlist HTTP/1.1
Host: www.opentosca.org

eventDateTime=2016-02-15&nodeInstanceID=42&eventMessage=booted
```

Die drei Informationen des Events sind *eventDateTime*, *nodeInstanceID* und *eventMessage*. Sie werden als Query-Parameter übermittelt. Die Namen der Parameter sind dieselben, wie sie bereits beim Vorgängerprojekt eingesetzt wurden, um eine Kompatibilität zu älteren Clients zu erhalten.

Eine weitere Besonderheit gegenüber einem Basic-Storage stellt eine Node Instance selbst dar. Es soll möglich sein, bestimmte Informationen zu einer solchen Node Instance über die REST-API abzufragen. Einige dieser Informationen können sich vergleichsweise schnell ändern. Um den Overhead beim Abfragen der momentanen Eigenschaften zu verringern, werden drei weitere Ressourcen hinzugefügt:

**Node Instance:** Diese Ressource ermöglicht den Abruf sämtlicher zu einer Node Instance gehörenden Eigenschaften. Dies sind insbesondere solche Informationen, die sich nicht (oft) ändern.

**Properties:** Hinter dieser Ressource verbirgt sich ausschließlich der „properties“-String der Node Instance. Ähnlich wie der Zustand, kann sich diese Zeichenkette oft ändern.

**State:** Unter dieser Ressource ist der aktuelle Zustand (und nur dieser) einer Node Instance abrufbar. Der Zustand kann sich oft ändern.

Die fertig modellierte OpenTOSCA-REST-Schnittstelle ist in Abbildung 4.7 zu sehen. Der OpenTOSCA-Teil wird dem *home document* angehängt, sodass sich ein Baum ergibt, der den Einstiegspunkt als Wurzel hat. Der Basic-Storage ist wegen der besseren Lesbarkeit in dieser Grafik nicht dargestellt. Im Gesamten hat die Schnittstelle damit die Form, wie sie in Abbildung 4.8 dargestellt ist. Weitere, zukünftig entwickelte Erweiterungen der API können in derselben Art und Weise angehängt werden. Um dabei die korrekte Referenzierung aufrecht zu erhalten und dadurch ein Traversieren des Baumes zu ermöglichen, muss eine solche Erweiterung natürlich im *home document* verankert werden.

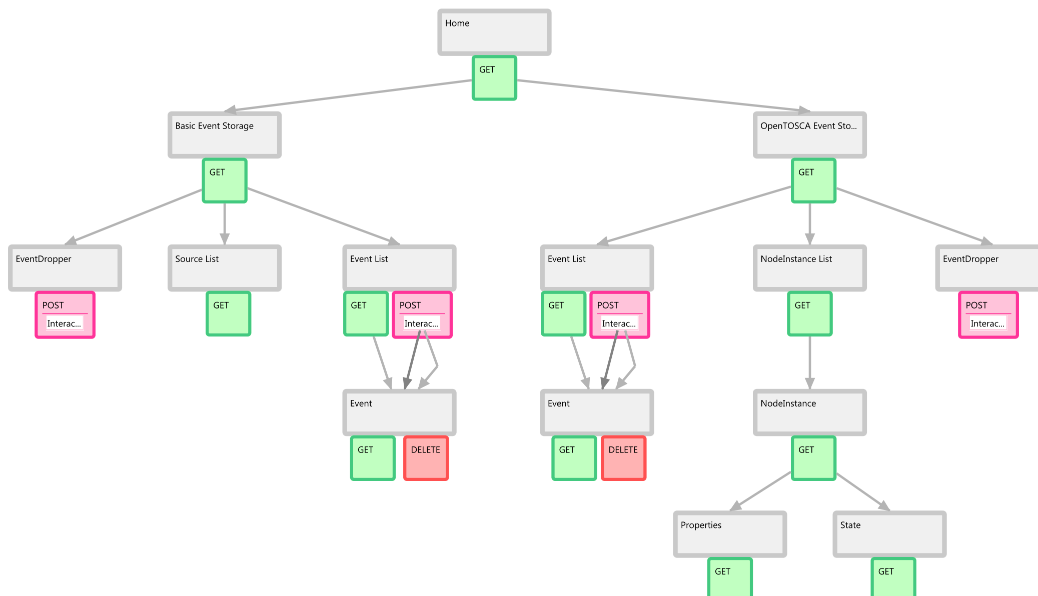


Abbildung 4.8: REST-Schnittstelle für den Basic Storage (links) und OpenTOSCA-Erweiterung (rechts)

Im Vergleich zum Basic Storage (Abbildung 4.3) kommen bei der OpenTOSCA-Erweiterung drei Use Cases hinzu, welche in Abbildung 4.9 dargestellt sind. Bei allen drei geht es um die Abfrage von Informationen zu einer Node Instance. Entweder werden alle verfügbaren Informationen erfragt, oder nur bestimmte Teile davon (der properties-String bzw. Zustand). Aus Sicht einer Eventquelle hat sich jedoch nichts geändert.

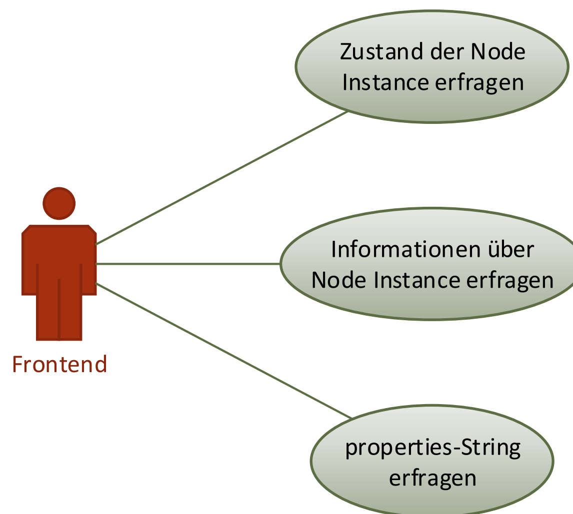


Abbildung 4.9: Zusätzliche Use Cases bei OpenTOSCA

### 4.4.2 OpenTOSCA-Komponenten

Ähnlich zu dem zuvor eingeführten Grundsystem, gibt es auch bei der OpenTOSCA-Erweiterung einen Teil hinter der REST-API. Dieser ist dem des Grundsystems sehr ähnlich. Es gibt ebenfalls eine Unterteilung in die drei Komponenten *Resources*, *Model* und *DAO*, deren Aufgaben exakt denen ihrer Geschwistern entsprechen: *Resources* stellt die Ressourcen der REST-API zur Verfügung und leitet die eingehenden Anfragen (Operationen auf die Ressourcen) an das *Data Access Object* weiter. Dieses verwaltet den Zugriff auf einen persistenten Speicher. Durch den modularen Aufbau kann dieser Speicher von dem des Grundsystems verschieden sein, er muss es aber nicht. Beide Komponenten greifen auf das im *Model* bereitgestellte Datenmodell zu, um ihre Aufgaben zu erfüllen. Die erreichte Kapselung ist damit dieselbe, wie sie bereits in Abbildung 4.6 dargestellt wurde.

Das Datenmodell nutzt die in Abschnitt 4.3.3 beschriebene Technik der Vererbung. Hierbei wird, wie in Abbildung 4.10 dargestellt, von dem Datenmodell des Basic-Storages abgeleitet. Die gemachten Änderungen bzw. Erweiterungen der Klassen sind nötig, um der Umbenennung von *Source* in *Node Instance* Rechnung zu tragen.

Bei dem Storage genügt für diese Umbenennung die Einführung zweier Methoden: `getNodeInstanceList()` und `setNodeInstanceList()`

Durch die in Abbildung 4.9 beschrieben, zusätzlichen Operationen wurde die Einbindung des sogenannten *OpenTOSCA Clients* nötig. Dieser ebenfalls an der Universität Stuttgart entwickelte REST-API Client stellt die Verbindung zu den OpenTOSCA-spezifischen Daten her. Er ermöglicht zum Beispiel die Abfrage des *properties*-Strings, der zu einer Node Instance gehört.

Um eine möglichst große Flexibilität bei der Speicherung der Events zu erreichen, wird für jeden Storage ein eigenes DAO erstellt (zu DAOs siehe Abschnitt 5.3). Bei der hier beschriebenen OpenTOSCA-Erweiterung übernimmt das DAO nicht nur das Lesen und Schreiben der Events von bzw. auf den persistenten Speicher, sondern stellt auch Methoden bereit, um die Informationen zu einer Node Instance durch den OpenTOSCA Client in Erfahrung zu bringen. Durch diese Abstraktionsschicht wäre es auch leicht möglich, den OpenTOSCA Client auszutauschen, ohne dabei weitere Teile (außer dem DAO) der OpenTOSCA-Erweiterung ändern zu müssen.

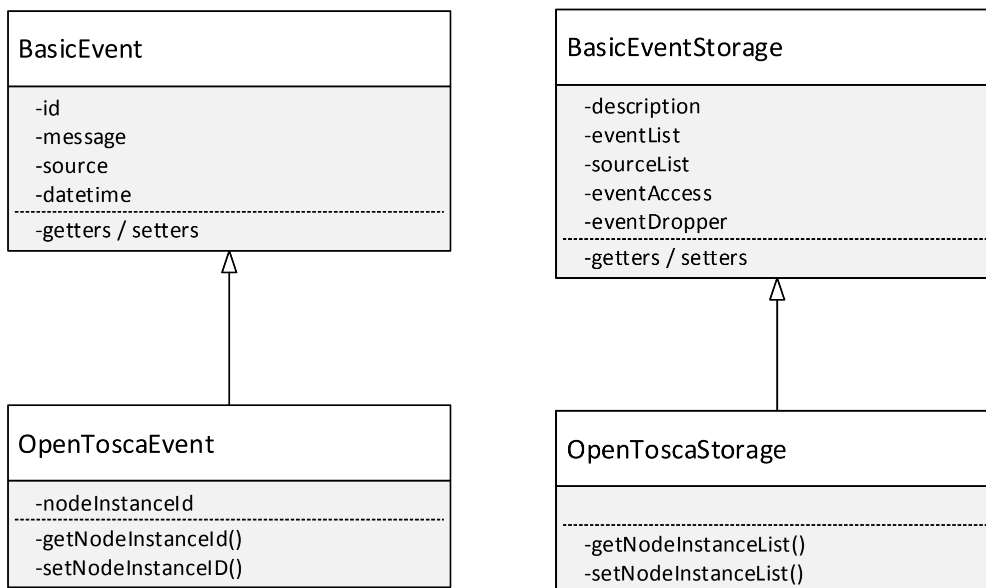


Abbildung 4.10: Vererbung des Datenmodells an die OpenTOSCA-Erweiterung

Im Gesamten kann eine Erweiterung also als „System im System“ gesehen werden. Sie wird in den vorhandenen Kontext eingebettet, ohne dabei in ihrer Funktion abhängig von den bereits existierenden Komponenten zu werden. Gleichzeitig können aber Vorteile wie etwa Wiederverwendbarkeit benutzt werden, um die Entwicklungskosten gering zu halten.

# 5 Implementierung

Dieser Abschnitt handelt von der Umsetzung des Entwurfs des Monitoring Backends. Dabei wurden verschiedene Technologien eingesetzt, die im Folgenden kurz vorgestellt werden. Anschließend wird auf wenige, wichtige Implementierungsdetails eingegangen. Zum Schluss werden die durchgeführten Tests behandelt.

## 5.1 Genutzte Technologien und Abhängigkeiten

Bei der Entwicklung kam eine Reihe unterschiedlicher Werkzeuge und Technologien zum Einsatz. Entwickelt wurde das Monitoring Backend als Java-Webanwendung, welche auf einem Tomcat-Server deployt wird. Um den Build- und Deploymentvorgang zu verwalten, wurde auf Apache Maven gesetzt. Die REST-Schnittstelle wurde mit JAX-RS (Java API for RESTful Web Services) und ihrer Implementierung Jersey umgesetzt. Um die Events abzuspeichern, wird auf die Java Persistence API (JPA) zurückgegriffen. Die Tests wurden mit SoapUI erstellt. Um sämtlichen Code zu verwalten, wurde Subversion eingesetzt.

### 5.1.1 Versionskontrolle

Für die Entwicklung des Monitoring Backends wurde als Versionsverwaltung *Apache Subversion* verwendet. Diese freie Software versioniert in einem zentralen Archiv, dem Repository. Dieses wird auf einem Server der Universität Stuttgart verwaltet. Ebenso wie für das Vorgängerprojekt wurde ein neues Unterprojekt angelegt, in das sämtliche dem Monitoring Backend zugehörige Dateien abgelegt wurden.

### 5.1.2 Entwicklungsumgebung

Für das Schreiben des eigentlichen Programmcodes wurde die Entwicklungsumgebung Apache Eclipse genutzt. Diese IDE (Integrated Development Environment) erleichtert den Umgang mit (Java-)Quellcode erheblich. Dies wird durch sehr hilfreiche Funktionen wie einer Autovervollständigung, Integration von Compilern oder Werkzeugen für das Refactoring von Code ermöglicht. Unzählige weitere Funktionen lassen sich durch Plugins nachrüsten. Eclipse ist open-source und frei verfügbar.

### 5.1.3 Maven

Apache Maven ist Tool, um den Build-Vorgang insbesondere von Java-Anwendungen zu verwalten. Dazu werden zwei Teilbereiche eines solchen Vorgangs betrachtet: Wie wird die Software gebaut und welche Abhängigkeiten hat sie. Diese Aspekte werden zentral konfiguriert. Ein Maven-Projekt kann dadurch sehr schnell auf und für unterschiedliche Plattformen gebaut werden. Die gute Integration von Maven in Eclipse ermöglicht eine komfortable Verwaltung. Unter anderem wurde mit einem Maven-Plugin so das Deployment des Monitoring Backends auf einen Tomcat-Server automatisiert.

### 5.1.4 Tomcat

Apache Tomcat ist ein Webserver, der die Servlet-Spezifikation (neben weiteren, wie etwa JSP oder Java EL) von Java-EE umsetzt. Dadurch ist es möglich, in Java entwickelte Webanwendungen auf Basis von Servlets auszuführen. Das Monitoring Backend wird als Servlet entwickelt und auf einen Tomcat-Server der Universität deployt.

### 5.1.5 JAX-RS / Jersey

Um die REST-Schnittstelle umzusetzen, wurde die Java API for RESTful Web Services (JAX-RS) genutzt. Ähnlich wie bei anderen Teiles von Java EE werden Annotations wie `@GET` oder `@PathParam` genutzt, um den Entwicklungsprozess von Web Service Clients und entsprechenden Endpunkten zu vereinfachen. JAX-RS wird unter anderem von Oracle Jersey implementiert, welches in diesem Projekt genutzt wurde.

### 5.1.6 Java Persistence API / Eclipselink / Derby

Die Java Persistence API (JPA) ist eine Spezifikation, die eine Abbildung von (Java-)Objekten auf relationale Datenbankeinträge definiert. Dies vereinfacht das Verwalten von Informationen aus einem objektorientierten Kontext in einer Datenbank erheblich. Teil der JPA ist die Java Persistence Query Language (JPQL), welche Abfragen sogenannter Entitäten erlaubt, die in der Datenbank gespeichert sind. Diese SQL ähnliche Sprache abstrahiert von den einzelnen Datenbanktabellen und ermöglicht eine einfache und komfortable Deserialisierung der Daten. Die Referenzimplementierung von JPA ist das quelloffene Eclipselink.
Für die (De-)Serialisierung selbst sind dabei drei Teile nötig: Die Schnittstelle (JPA), ein sogenannter „persistence provider“, also die Implementierung der Schnittstelle (Eclipselink) sowie die eigentliche Datenbank, der “persistence driver“.

Apache Derby, ein relationales Datenbank-Mangement-System, ist der persistence driver, der beim Monitoring Backend eingesetzt wird.

### 5.1.7 SoapUI / Tests

SoapUI ist ein in der Basisversion frei verfügbares Werkzeug für das Testen von Software. Mit ihm können (neben vielen anderen) REST-Webdienste getestet werden. Durch die Implementierung in Java ist es plattformunabhängig. Die damit durchgeführten Tests des Monitoring Backends betrachten entsprechend die Funktionalität der Schnittstelle nach außen hin und weniger die dahinter verborgenen Methoden.

### 5.1.8 Swagger

Swagger (`http://swagger.io/`) ist zunächst eine Spezifikation, um REST-Dienste zu definieren. Sie erleichtert die Erstellung von Client und Server, die über eine REST-API kommunizieren, indem Code, Dokumentation und Tests generiert werden können.
Im Rahmen des Monitoring Backends wurde Swagger genutzt, um bei der Entwicklung die REST-API nach dem Deployment auf den Tomcat-Server zu repräsentieren. Diese Repräsentation erleichtert das Überprüfen und schnelle Testen der API. Es wurden zwei Komponenten von Swagger genutzt: Mit Hilfe von Annotationen wird eine Beschreibung der REST-Schnittstelle generiert, welche später von einem Frontend eingelesen wird, um dann mit der API zu interagieren. Dieses Frontend wurde als eigenständiges Projekt verwaltet, um die Nutzung vom Monitoring Backend unabhängig zu machen.

## 5.2 REST-API

Die REST-Schnittstelle wird über JAX-RS-Annotationen umgesetzt. Die Implementierung Jersey erlaubt in der Konfiguration die Angabe von Java-Paketen, die nach entsprechenden Annotationen gescannt werden sollen. Im Hinblick auf die Trennung einzelner Storages gibt es entsprechend pro Storage ein Paket „resources“, in welchem sich die annotierten Klassen befinden. Dies soll am Beispiel der Ressource verdeutlicht werden, die einzelne Events verwaltet:

**Listing 7** JAX-RS Annotationen bei der REST-API

```
@Path("/eventstorages/basic/event/{eventID}")
@Produces(MediaType.APPLICATION_JSON)
public class BasicEventResource {

  @GET
  public BasicEvent doGet(@PathParam("eventID") String eventID) {
    [...]
    BasicEvent result = dao.getEvent(Integer.parseInt(eventID));
    return result;
  }
  [...]
}
```

Die in Listing 7 verkürzt dargestellte Klasse ist mit drei Annotationen versehen:

- `@Path` erlaubt die Angabe eines URI-Schemas, das zu einer Ressource gehört. In diesem Beispiel ist dies `/eventstorages/basic/event/{eventID}`. Der durch geschweifte Klammern eingeschlossene Teil `eventID` entspricht einer später verwendeten Variable, die das jeweilige Event identifiziert.
- `@Produces` gibt an, welcher MIME-Type bei der Kommunikation verwendet werden soll. Jersey kann die Konvertierung verschiedener Datentypen, insbesondere auch von Klassenfeldern, automatisiert durchführen. Dadurch wird trotz des Rückgabetyps `BasicEvent` der Methode später ein JSON-String, der das Objekt repräsentiert, an den Client zurückgeschickt.
- `@GET` markiert die darauffolgende Methode, sodass diese bei HTTP-GET-Requests auf die durch `@Path` zuvor identifizierte Ressource aufgerufen wird.

Die Methode selbst übernimmt die aus dem Pfad extrahierte `eventID` als Parameter, leitet die Anfrage an das DAO weiter und gibt das Ergebnis anschließend zurück.

## 5.3 Persistenz

Die von der REST-API empfangenen Events werden persistent gespeichert. Dies wird durch ein Data Access Object (DAO) bewerkstelligt. Dazu wurde eine Schnittstelle definiert, welche generische Operationen anbietet, etwa um Events zu speichern, abzurufen und zu löschen. Die Implementierung dieser Schnittstelle bestimmt, auf welchem Speicher die Daten letztlich abgelegt werden. Bei einer Speicherung in eine Datenbank etwa übernimmt die Implementierung des DAOs die Verbindungsverwaltung zum DBMS sowie die eigentlichen Abfragen (Queries). Dieser Ansatz erlaubt das unkomplizierte Austauschen des dahinterliegenden

Speichers, indem eine entsprechend andere Implementierung geschrieben wird. So könnte eine weitere Umsetzung die Events auf das lokale Dateisystem speichern (Abbildung 5.1).

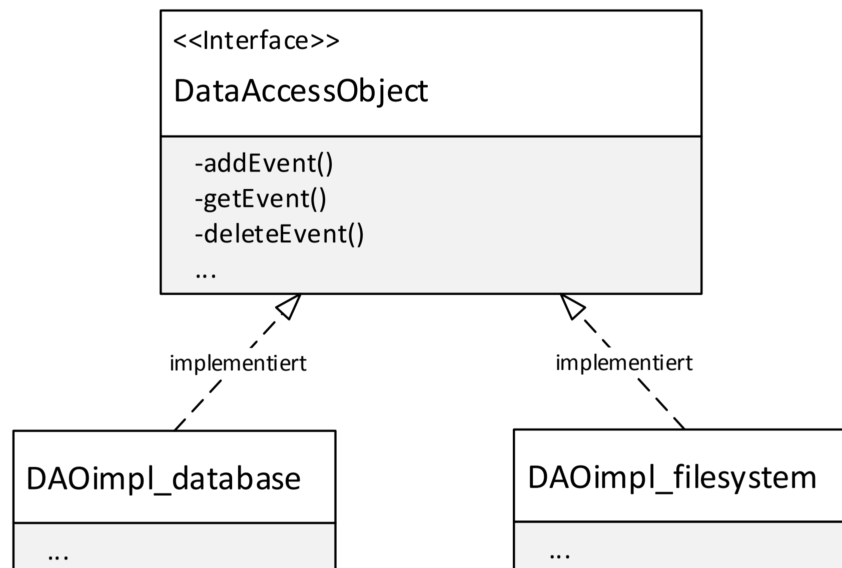


Abbildung 5.1: Speicherung der Events: Interface und Implementierung des DAOs

Um Events mit JPA speichern zu können, müssen sogenannte *Entities* über Annotations festgelegt werden. Diese Entities werden dann in einer Datenbank abgelegt und später wieder abgefragt.
Listing 8 verdeutlicht diese Handhabung. Die Klasse `BasicEvent` wird mit `@Entity` zu einer Entity deklariert, die mittels JPA/Derby verwaltet werden soll. Dies führt im Hintergrund zum Anlegen entsprechender Tabellen und Referenzen (Wie sie auftreten können, wenn etwa eine Klassenvererbung existiert und gespeichert werden soll.).
Mit `@ID` wird das Feld einer Klasse bestimmt, welches als Primärschlüssel im Datenbankkontext dienen soll. `@GeneratedValue` gibt an, dass das Datenbank-System sich um die Verwaltung dieses Schlüssels kümmern soll, indem die entsprechenden Werte automatisch generiert werden.
Die letzte Annotation `@Temporal` ermöglicht die Angabe eines bestimmten Zeit-Typs. Dies führt zur korrekten (De-)Serialisierung von Datumsangaben. Werden solche Zeitangaben korrekt gespeichert, erleichtert dies auch die Abfrage von Entitäten aus der Datenbank. So ist damit leicht möglich, in der JPQL Filter einzubauen, die ein bestimmtes Datum berücksichtigen. Ein weiterer Nutzen ergibt sich bei der Sortierung nach Datum.

**Listing 8** JPA-Annotationen zur persistenten Speicherung von Events

```
@Entity
public class BasicEvent {
  private int id;
  private Date datetime;
[...]

  @Id
  @GeneratedValue
  public int getId() {
    return id;
  }

  @Temporal(TemporalType.TIMESTAMP)
  public Date getDatetime() {
    return datetime;
  }
[...]
}
```

## 5.4 Tests

Das Testen von Software ermöglicht es, bestimmte Verhaltensweisen zu überprüfen. Dabei jedoch nie die Abwesenheit von Fehlern gezeigt werden. Dieser Abschnitt beschäftigt sich nicht mit jedem einzelnen Testfall, sondern soll zeigen, welche Methode hinter dem Testen des Monitoring Backends liegt.
Da es sich um eine Web-Anwendung handelt, wurde die REST-Schnittstelle ausführlich getestet. Hierzu wurden hauptsächlich Use-Cases ähnlich denen in Abschnitt 4.2 vorgestellten verwendet. Verbindet man solche Use-Cases miteinander, können sehr große Teile des Systems sehr einfach getestet werden, es wird eine große Abdeckung erreicht. Ein Beispiel hierfür ist Abrufen der bereits vorhandenen Events, das Abspeichern eines neuen Events und das anschließende, erneute Abfragen der vorhandenen Events. Dabei werden viele unterschiedliche Teile des Systems benötigt und diese auch getestet. In Abbildung 5.2 ist der oben genannte Fall dargestellt. Hierbei werden werden, ähnlich zu Abbildung 4.6, die Komponenten des Grundsystems *Resources*, *DAO* und der *persistente* Speicher genutzt.

Durch die Verknüpfung mehrerer Operationen können Abhängigkeiten berücksichtigt werden. Im oben genannten Beispiel muss das Event `E` nach dem Speichern in der dann neu abgerufenen Liste der Events auftauchen. Dabei muss es exakt rekonstruiert werden: Es darf etwa kein anderes Datum oder eine andere Nachricht enthalten.

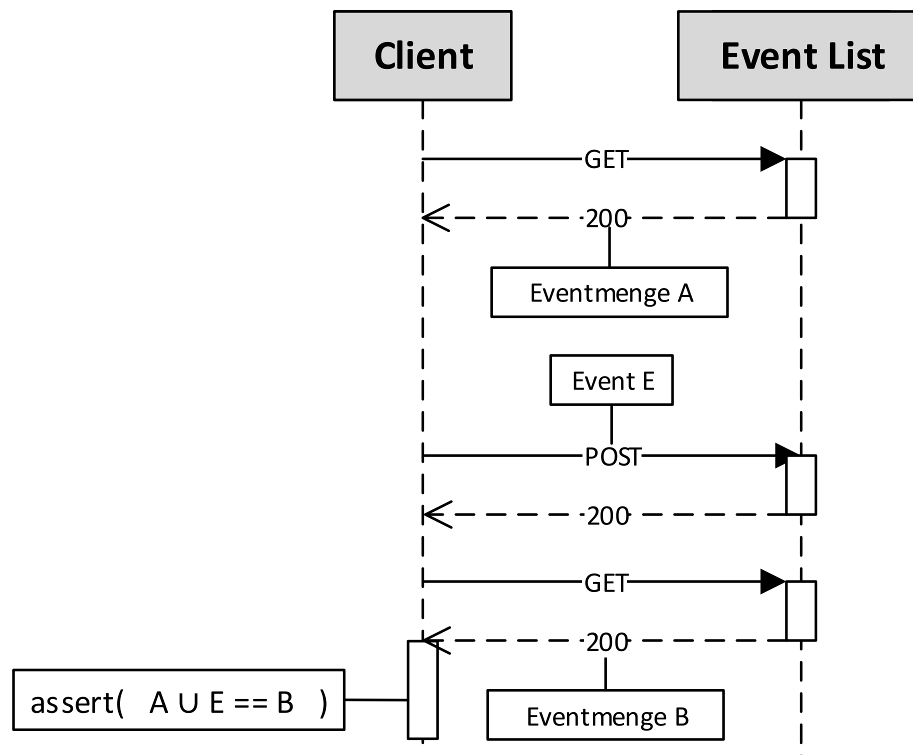


Abbildung 5.2: Testfall: Events abfragen und Event speichern

In einer ähnlichen Weise wurden die vielen anderen Testfälle erstellt. Dabei wurden unter anderem die verschiedenen HTTP-Operationen auf den Ressourcen, die Sortierung und das Löschen von Events, sowie die Informationsabfrage zu den Storages getestet.

# 6 Zusammenfassung und Ausblick

Unter Representaional state transfer (REST) versteht man einen Architekturstil für verteilte Systeme. Dabei werden verschiedene Einschränkungen (Constraints) definiert, die eine Anwendung einhalten muss. Systeme, die nach den REST-Prinzipien entwickelt wurden, können wachsen, ohne Eigenschaften wie Performanz, Skalierbarkeit, Modifizierbarkeit oder Portabilität zu verlieren. Dies ist ein großer Vorteil unter anderem, wenn Anwendungen zunächst im Kleinen beginnen, man sich aber die Möglichkeit offen halten will, später Erweiterungen hinzuzufügen.
Bei der Provisionierung von Cloud-Anwendungen in der OpenTOSCA-Laufzeitumgebung treten Prozesse, über welche man gerne genau Bescheid wüsste, in welchem Zustand sie sich befinden oder ob Fehler aufgetreten sind.
Ein Vorprojekt hatte zum Ziel, während dieser Prozesse geschickte Events entgegenzunehmen, zu speichern und für die spätere Verwendung abrufbar zu machen. Im Rahmen dieser Bachelorarbeit wurde dieses Vorgängerprojekt analysiert und seine Schwachstellen, im Besonderen in Hinblick auf Erweiterbarkeit und Modularität, herausgestellt.
Anschließend wurde ein Entwurf für ein verbessertes System erarbeitet. Es sollte die gleiche Funktionalität bieten, also Events entgegennehmen, persistent speichern und zur Verfügung stellen, dabei jedoch die Probleme des Vorprojektes vermeiden. Das Augenmerk bei der Entwicklung dieses Monitoring Backends lag auf der Erweiterbarkeit seiner Komponenten. Hierzu wurden sogenannte Storages eingeführt, die eine möglichst unabhängige Verwaltung verschiedener Eventtypen ermöglichen. Durch Techniken wie Vererbung wurde eine Hierarchie bei den Storages und Events eingeführt, um eine hohe Flexibilität zu erlauben. Dadurch wird es möglich, sowohl generische Clients für die Kommunikation mit jedem Storage zu erstellen, als auch sehr spezialisierte Versionen, die storagespezifische Details verarbeiten können. Der Entwurf eines Storages gliedert sich in zwei Teile: Die REST-API als Bindeglied zwischen Client und Server, und das dahinterstehende System zur Speicherung besagter Events. Ferner wurde in diesem Teil der Arbeit auf den Entwurf einer Erweiterung eingegangen. Dafür wurde ein OpenTOSCA-Storage beispielhaft entworfen, indem seine REST-Schnittstelle und das dahinterliegende System erarbeitet wurden.
Daraufhin wurde die Implementierung des zuvor erstellten Entwurfs beschrieben. Es wurden die verwendeten Technologien eingeführt und auf Schlüsselmechanismen wie die Bereitstellung der REST-API oder persistente Speicherung der Events eingegangen. Abschließend wurden die durchgeführten Tests beschrieben.
Durch die Konzentration auf Merkmale wie Erweiterbarkeit und Skalierbarkeit ist es leicht möglich, für ähnliche Problemstellungen weitere Storages zu entwerfen und zu Implementieren. Die verhältnismäßig geringe Komplexität von OpenTOSCA-Events (bzw. Storages) zeigt dabei noch nicht das ganze Potential des Monitoring Backends. Je nach Grad der Integration von entsprechenden Clients, die Events schicken können, bietet sich das Backend auch an, um eine generische, zentrale Log-Funktionalität anzubieten.

# Literatur


[1] Roy Thomas Fielding. „Architectural styles and the design of network-based software architectures“. Diss. University of California, Irvine, 2000.

[2] Martin Fowler. *Richardson Maturity Model*. URL: http://martinfowler.com/articles/richardsonMaturityModel.html (besucht am 04. 05. 2016).

[3] Longshine Technologie Europe GmbH. *OSI-7-Schichtmodell*. URL: http://forum.longshine.de/phpbb2/bilder/ur2/osi2.gif (besucht am 08. 05. 2016).

[4] Florian Haupt, Frank Leymann und Cesare Pautasso. „A conversation based approach for modeling REST APIs“. In: *12th Working IEEE / IFIP Conference on Software Architecture - WICSA 2015*. IEEE Computer Society, 2015.

[5] Florian Haupt u. a. „A Model-Driven Approach for REST Compliant Services“. In: *Proceedings of the IEEE International Conference on Web Services (ICWS 2014)*. IEEE, 2014, S. 129 –136. DOI: 10.1109/ICWS.2014.30.

[6] IETF. *Hypertext Transfer Protocol – HTTP/1.1*. URL: https://www.ietf.org/rfc/rfc2616.txt (besucht am 14. 04. 2016).

[7] IETF. *Hypertext Transfer Protocol (HTTP/1.1): Caching*. URL: https://tools.ietf.org/html/rfc7234 (besucht am 04. 05. 2016).

[8] IETF. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. URL: https://tools.ietf.org/html/rfc7230 (besucht am 11. 04. 2016).

[9] IETF. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. URL: https://tools.ietf.org/html/rfc7231 (besucht am 12. 04. 2016).

[10] IETF. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. URL: https://www.ietf.org/rfc/rfc2045.txt (besucht am 04. 05. 2016).

[11] IETF. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. URL: https://www.ietf.org/rfc/rfc2046.txt (besucht am 04. 05. 2016).

[12] IETF. *PATCH Method for HTTP*. URL: https://tools.ietf.org/html/rfc5789 (besucht am 14. 04. 2016).

[13] IETF. *Uniform Resource Identifier (URI): Generic Syntax*. URL: https://tools.ietf.org/html/rfc3986 (besucht am 11. 04. 2016).

[14] OASIS. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (besucht am 18. 04. 2016).

[15] Leonard Richardson und Sam Ruby. *Restful Web Services*. 1. Aufl. O’Reilly, 2007. ISBN: 9780596529260.

[16] Stefan Tilkov. *REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien*. 2. Aufl. dpunkt Verlag, 2009. ISBN: 9783898645836.

[17] Oliver Kopp Frank Leymann Tobias Binz Uwe Breitenbücher. *TOSCA - Topology and Orchestration Specification for Cloud Applications*. URL: `http://install.opentosca.org/documentation/Documents/Presentation_TOSCA.pdf` (besucht am 08. 05. 2016).

[18] J ham3 (commonswiki). *Network Community Structure (Creative Commons license)*. URL: `https://upload.wikimedia.org/wikipedia/commons/f/f4/Network_Community_Structure.svg` (besucht am 08. 05. 2016).

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