



**Masterarbeit**

# Entwurf und Integration eines Frameworks zur Einhaltung nutzerdefinierter Policies in OpenStack

**Design and Integration of a Framework Enforcing User-Defined Policies in OpenStack**

Matthias Bastian

**Betreuung**
Prof. Dr. rer. nat. habil. Andreas Polze, Max Plauth, M. Sc.
*Fachgebiet für Betriebssysteme und Middleware*

Hasso-Plattner-Institut an der Universität Potsdam

24. Januar 2017

**Zusammenfassung**

Die Popularität von Cloud-Computing hat in den letzten Jahren stark zugenommen. Die Nutzung von Cloud-Ressourcen ist sowohl für Privatpersonen als auch für Unternehmen zunehmend zur Normalität geworden. Zugleich wirft dies Fragen hinsichtlich des Datenschutzes auf, da die in Cloud-Umgebungen befindlichen Daten außerhalb des Kontrollbereiches ihres Eigentümers sind. Cloud-Provider veröffentlichen zwar ihre Datenschutzbestimmungen, allerdings ist ein vollständiges Einverständnis Voraussetzung zur Nutzung der Ressourcen. Diese Arbeit geht von einem umgedrehten Ansatz aus: Nutzer können ihre Anforderungen detailliert mithilfe von Policies ausdrücken, denen die Cloud-Anbieter zustimmen sollen.

Zur Behandlung vielfältiger Policies müssen Cloud-Plattformen umfassend erweitert werden. Diese Arbeit stellt daher eine prototypische Umsetzung für OpenStack vor. Zunächst werden grundlegende Entwurfsentscheidungen beleuchtet. Dazu gehört das Ziel, die Unterstützung von Policies so komfortabel wie möglich zu gestalten, damit die praktische Umsetzung nicht durch den notwendigen Implementierungsaufwand beeinträchtigt wird. Anschließend werden die elementaren Anpassungen an OpenStack beschrieben, unter anderem die Speicherung und Abfrage von Policies.

Es folgt die Erläuterung einer Middleware, die den einzelnen OpenStack-Diensten die Policy bereitstellt. Dazu stattet sie jede bei einem Dienst eingehende Anfrage mit der Policy des anfragenden Nutzer aus, wodurch sich bereits der Erhalt einer Policy aus Sicht der Dienste einfach gestaltet.

Schwerpunkte der Arbeit sind der Entwurf und die Umsetzung eines Frameworks, das die Erweiterung der Dienste um Unterstützung von Policies ermöglicht. Das Framework bietet eine Plugin-Architektur, bei welcher die Erweiterungen die Unterstützung einzelner Policies umsetzen. Die Verlagerung der Logik in die Erweiterungen reduziert die benötigten Anpassungen an den eigentlichen OpenStack-Diensten auf ein Minimum.

Zur Veranschaulichung des Frameworks wurden drei Erweiterungen beispielhaft umgesetzt, bei denen die Policy die Verschlüsselung virtueller Datenträger vorschreibt, die Nutzung der Cloud-Ressourcen auf ausgewählte geographische Standorte beschränkt und eine replizierte Ausführung bestimmter virtueller Maschinen gewährleistet. Die Arbeit geht ebenso auf Features und Interna des Frameworks ein und beschreibt abschließend die erfolgte Erweiterung von OpenStacks Web-Dashboard zur dortigen Verwaltung von Policies sowie die dabei aufgetretenen Herausforderungen.

# Inhaltsverzeichnis


**1 Einleitung** **1**

**2 Verwandte Arbeiten** **4**
2.1 Policy-Sprachen . . . . . 4
2.1.1 XACML . . . . . 4
2.1.2 CPPL . . . . . 4
2.2 OpenStack . . . . . 5
2.3 Policy-Ansätze in OpenStack . . . . . 5
2.3.1 oslo.policy . . . . . 5
2.3.2 Swift Storage Policies . . . . . 7
2.3.3 Policy-based Scheduling . . . . . 7
2.3.4 Group Based Policy . . . . . 7
2.3.5 Congress . . . . . 8

**3 Grundlegende Entwurfsentscheidungen** **10**

**4 Speicherung von Policies in Keystone** **13**
4.1 Entwicklungsumgebung für Keystone . . . . . 14
4.2 Anpassungen am Datenmodell von Keystone . . . . . 16
4.3 Abfragen und Setzen einer Nutzer-Policy . . . . . 17

**5 Kommunikation mit Keystone** **19**
5.1 Anpassungen an python-keystoneclient . . . . . 19
5.2 Entwicklungsumgebung für python-keystoneclient . . . . . 19

**6 Aufbau und Konzepte einer Middleware zur Bereitstellung von Policies** **21**
6.1 Middlewares in OpenStack . . . . . 21
6.2 Beispiel: keystonemiddleware . . . . . 22
6.3 policymiddleware . . . . . 23
6.3.1 Middleware kontra Bibliothek . . . . . 23
6.3.2 Konfiguration . . . . . 25
6.3.3 Integration in oslo.middleware . . . . . 26
6.3.4 Entwicklungsumgebung für oslo.middleware . . . . . 26

**7 Das policyextension-Framework** **28**
7.1 Motivation . . . . . 28
7.2 Entwicklungsumgebung für policyextension . . . . . 30

7.3 Policy-Beispiel 1: Festplattenverschlüsselung . . . 30
7.3.1 Entwicklungsumgebung für Cinder . . . 31
7.3.2 Festplattenverschlüsselung in OpenStack . . . 33
7.3.3 Umsetzung der Policy . . . 34
7.4 Policy-Beispiel 2: Einschränkung der Availability Zones . . . 37
7.4.1 Segregation physischer Infrastruktur . . . 37
7.4.2 Entwicklungsumgebung für Nova . . . 40
7.4.3 Umsetzung der Policy . . . 42
7.5 Policy-Beispiel 3: Replikation . . . 44
7.5.1 Replikation von Anwendungen oder von VMs? . . . 44
7.5.2 Umsetzung der Policy . . . 48
7.5.3 Evaluierung . . . 53
7.5.4 Entwicklungsumgebung . . . 54
7.6 Unabhängigkeit vom Format der Policy durch PolicyFormat . . . 54
7.7 Features und Interna des policyextension-Frameworks . . . 56
7.7.1 Ausführungsreihenfolge mittels Prioritäten . . . 57
7.7.2 Fehlerklassen . . . 57
7.7.3 Versionsrestriktion . . . 58
7.7.4 Ausführung der PolicyExtensions mittels Monkey Patching . . . 59
7.7.5 Bereitstellung der Policy durch Inspektion des Aufrufstapels . . . 59
7.7.6 Bereitstellung der Argumente der Originalfunktion . . . 60
7.7.7 Dekorierer und Python-Versionskompatibilität . . . 61
7.8 Diskussion: Stärken und Schwächen . . . 63

**8 Integration ins Web-Dashboard** **67**
8.1 Abfrage und Aktualisierung von Policies . . . 67
8.2 Fehlerbehandlung . . . 68
8.3 Entwicklungsumgebung für Horizon . . . 70

**9 DevStack als Entwicklungsumgebung** **72**

**10 Zusammenfassung und Ausblick** **76**

**Literaturverzeichnis** **81**

# 1 Einleitung

*Cloud-Computing* hat sich in den letzten Jahren zu einem weit verbreiteten Oberbegriff entwickelt. Allen Teilbereichen, wie zum Beispiel Software as a Service (SaaS), Platform as a Service (PaaS) und Infrastructure as a Service (IaaS) ist gemein, dass letztlich Speicher- oder Rechenressourcen entfernter Rechner genutzt werden [19].

Während zwar die in einer Cloud angebotenen Ressourcen die eigenen, lokal vorhandenen Möglichkeiten in aller Regel bei Weitem übersteigen, ist dem Cloud-Computing inhärent, dass Daten zum Cloud-Anbieter übertragen und für gewöhnlich auch dort gespeichert werden. Für Unternehmen stellt deshalb die Verbindung von Cloud-Computing und Datenschutz besonders im Hinblick auf Kundendaten eine Herausforderung dar. Durch die NSA-Affäre wurde einer breiten Öffentlichkeit bewusst, welches Missbrauchspotential große Datensammlungen bergen. Gerade beim Cloud-Computing nützt das Vertrauen in den Cloud-Anbieter jedoch nichts, wenn dieser aufgrund rechtlicher Rahmenbedingungen verpflichtet werden kann, Daten an staatliche Stellen herauszugeben.

Mit Amazon, Microsoft, IBM und Google wird der Cloud-Computing-Markt von US-Unternehmen dominiert [40]. Durch die NSA-Affäre mussten die Anbieter fürchten, ihren Kunden keine Sicherheit der Daten gewährleisten zu können. Trotz besserer Absicherung der Infrastruktur und Eröffnung neuer Rechenzentren in Regionen außerhalb von US-Gebiet haben Öffentlichkeit und Unternehmen noch immer Bedenken im Hinblick auf Cloud-Nutzung, insbesondere bei US-Unternehmen: Mehr als die Hälfte deutscher Unternehmen mit mehr als 1000 Mitarbeitern hat Vorbehalte gegenüber Cloud-Diensten und fürchtet insbesondere unberechtigten Zugriff [15]. Zwar ist die Cloud-Nutzung unter deutschen Unternehmen von 44% im Jahr 2015 auf 54% im Jahr 2016 gestiegen, zugleich fordern aber 76% der Kunden, Rechenzentren in Deutschland anzubieten, um das Risiko des unberechtigten Zugriffs auf sensible Unternehmensdaten zu verringern [17].

Um das Vertrauen in ihre Dienste wiederherzustellen, kommen US-Anbieter diesem Wunsch nach und stellen Rechenzentren in Europa und Deutschland bereit. Dropbox beispielsweise bietet Unternehmenskunden die Speicherung ihrer Daten in Deutschland an [11]. Dieser Ansatz scheint zunächst erfolgreich, so ist bei Amazon Web Services (AWS) Frankfurt/-Main die am stärksten wachsende Region [10]. Anderseits vertritt die US-Regierung die Ansicht, ein US-Unternehmen müsse Daten auf Anfrage herausgeben, selbst wenn sie außerhalb des US-Territoriums gespeichert sind. Bezüglich dieser Frage befanden sich Microsoft und die US-Regierung seit 2014 in einem Rechtsstreit, bei dem es um die Herausgabe von in Irland gespeicherten E-Mails eines Nutzer ging. Im Juli 2016 urteilte ein US-Bundesberufungsgericht im Sinne Microsofts, dass US-Behörden keine Datenherausgabe erzwingen könnten. Die US-Staatsanwaltschaft gibt sich damit jedoch nicht zufrieden und versucht den Fall neu aufrollen zu lassen [39].

Um einen Zugriff auszuschließen, stellt Microsoft Geschäftskunden seine Cloud-Ressourcen in der EU auch über das Unternehmen T-Systems, das als Treuhänder fungiert, zur Verfügung. Bei diesem Modell soll Microsoft selbst keinen Zugriff auf Kundendaten haben und diese somit nicht US-Behörden zur Verfügung stellen können [1].

Europäische Cloud-Anbieter nutzen ihre Herkunft als Werbeargument, die Telekom beispielsweise bewirbt ihre *Open Telekom Cloud* als "sichere europäische Cloud" [6]. Die in *Cloud Infrastructure Services Providers in Europe (Cispe)*[1] zusammengeschlossenen Cloud-Infrastrukturanbieter versuchen das Vertrauen der Kunden durch eine Selbstverpflichtung zu gewinnen.

Einen anderen Ansatz verfolgt das Bundesamt für Sicherheit in der Informationstechnik (BSI): Es veröffentlicht einen Anforderungskatalog für Cloud-Anbieter, anhand dessen zum einen Kunden die Anbieter beurteilen können sollen und zum anderen die Anbieter sich auf eine Zertifizierung vorbereiten können [41]. Das Bundesministerium für Wirtschaft und Energie (BMWi) geht mit seiner Zertifizierung und anschließenden Verleihung des Labels *Trusted Cloud*[2] ähnlich vor.

Die Europäische Union (EU) hat die Bedeutung von Cloud-Computing und die Dominanz durch US-Unternehmen verstanden. Im Rahmen des *Horizon2020*-Programms[3] fördert die Europäische Kommission daher das *Scalable and Secure Infrastructures for Cloud Operations (SSICLOPS)*-Projekt. Darin haben sich eine Vielzahl europäischer Universitäten und Unternehmen zusammengetan, um unter anderem Techniken zur Föderation von Cloud-Infrastrukturen zu entwickeln, in denen die durch Software Defined Networking (SDN) erstellten Netzwerke über Cloud-Grenzen hinweg verfügbar sind. Die Zusammenschaltung mehrerer Clouds unterschiedlicher Anbieter ermöglicht für Kunden zwar eine hohe Flexibilität und neue Möglichkeiten, erschwert aber die Auswahl eines passenden Anbieters wegen verschiedener Datenschutzbestimmungen sogar noch. So kann es in einem Cloud-Verbund Anbieter geben, die unter eine gewünschte Jurisdiktion fallen, andere hingegen könnten aufgrund ihres geographischen Standortes ausgeschlossen werden müssen. In einem solchen Szenario stellt sich die Frage, ob damit der gesamte Cloud-Verbund nicht infrage kommt oder ob spezifizierbar ist, welche Anbieter des Verbunds in Anspruch genommen werden können.

Diese Arbeit beschäftigt sich mit der Integration solcher vom Nutzer vorgegebenen Spezifikationen, genannt *Policies*, in eine verteilte Cloud-Computing-Plattform. Die Komplexität der Plattform stellt eine Herausforderung dar, sodass ein Schwerpunkt der Arbeit die Frage behandelt, wie die Integration dennoch vorgenommen werden kann, zugleich aber möglichst wenig Änderungen an der Plattform benötigt. Neben der architektonischen Fragestellung ist die Vielseitigkeit möglicher Nutzeranforderungen eine weitere Herausforderung: Policies können weit komplizierter sein, als nur einzelne Anbieter auszuschließen oder die Lage von Rechenzentren vorzuschreiben. Denkbar sind beispielsweise Vorgaben zur Datenverschlüsselung oder zur automatischen Löschung von Daten nach einer gewissen Zeit. Die Unterstützung einer Cloud-Umgebung für vom Nutzer vorgegebene Policies stellt darüber hinaus einen grundlegenden Prinzipienwechsel dar: Bislang hat der Nutzer zwar die Aus-

[1] https://cispe.net/

[2] https://www.trusted-cloud.de/

[3] http://ec.europa.eu/programmes/horizon2020/

wahl zwischen verschiedenen Providern. Deren Umgang mit Nutzerdaten ist für den Kunden aber nicht verhandelbar. Ist dieser nicht vollumfänglich einverstanden, bleibt ihm nur die Suche nach einem passenderen Angebot der Konkurrenz, falls er nicht trotzdem zuzustimmen bereit ist. Vom Nutzer angegebene Policies hingegen schreiben der Cloud vor, wie sie ihr Verhalten anpassen muss, um den Nutzeranforderungen gerecht zu werden. Unter diesen Umständen lassen sich auch komplizierte Compliance-Vorgaben ausdrücken sowie Datenschutzanforderungen besser mit Cloud-Angeboten in Einklang bringen.

Diese Arbeit betrachtet im Folgenden thematisch verwandte Arbeiten und stellt anschließend die grundlegenden Erweiterungen einer Cloud-Plattform vor, um die Verwaltung von Policies zu ermöglichen. Es folgt die Beschreibung eines Frameworks, das mit minimalem Aufwand die Unterstützung von Policies in der gewählten Cloud-Plattform ermöglicht. Zur Veranschaulichung des Frameworks werden drei Beispiele erläutert, die jeweils für die Umsetzung einer Policy sorgen.

# 2 Verwandte Arbeiten

Dieser Abschnitt geht zunächst auf Möglichkeiten zum Ausdruck von Policies, sogenannte *Policy-Sprachen*, ein, stellt anschließend eine Open Source Cloud-Plattform vor, die im Rahmen dieser Arbeit um Policy-Unterstützung ergänzt wird und zeigt, welche Ansätze für Policies dort bereits existieren.

## 2.1 Policy-Sprachen

Policies sind natürlich-sprachliche Regelungen, die technisch automatisiert um- und durchgesetzt werden sollen. Um diese Regelungen maschinenlesbar zu gestalten, existiert eine Vielzahl von Policy-Sprachen, mit denen sich Policies ausdrücken lassen.

### 2.1.1 XACML

eXtensible Access Control Markup Language (XACML) [13] ist die vorwiegend eingesetzte Policy-Sprache bei der Spezifikation von Zugriffsrechten. Die Policies werden mittels Extensible Markup Language (XML) dargestellt, womit sie sowohl für Maschinen als auch für Menschen lesbar sind – für letztere allerdings recht aufwendig, wenn man XML beispielsweise mit der Auszeichnungssprache *YAML Ain't Markup Language (YAML)* [2] vergleicht.

### 2.1.2 CPPL

Im Rahmen des SSICLOPS-Projektes wurden an der RWTH Aachen neun bestehende Policy-Sprachen, darunter XACML, auf ihre Tauglichkeit hinsichtlich eines Einsatzes in Cloud-Umgebungen geprüft [8]. Dazu wurde eine Reihe von Kernanforderungen an eine geeignete Policy-Sprache aufgestellt:

- Geringer Speicherbedarf: Wenn Policies in hohem Maße zusätzliche Ressourcen benötigen, ist ihr Einsatz für Cloud-Anbieter unattraktiv. Sie sollen deshalb nur wenig Speicher benötigen.
- Menschenlesbarkeit und Verständlichkeit: Nutzern soll die Verfassung von Policies leicht fallen.
- Erkennung von Widersprüchlichkeiten: Die Policy-Sprache soll widersprüchliche Policies erkennen, zum Beispiel könnte eine Policy US-Rechenzentren ausschließen, eine andere jedoch zeitgleich Backups in einem New Yorker Rechenzentrum verlangen.
- Performance: Die Policies müssen womöglich oft geprüft werden, daher soll ihre Auswertung schnell geschehen können.

- Ausdrucksstärke: Policies sollen eine Vielzahl von Regeln ausdrücken können, beispielsweise Speicherorte, Verschlüsselungsanforderungen, automatische Löschungen, erforderliche Replikationsraten oder Benachrichtigungen bei Zugriff durch Dritte.
- Erweiterbarkeit: Um für zukünftige Anwendungsfälle geeignet zu sein, soll die Sprache erweiterbar sein.
- Übereinstimmungsprüfung: Die Nutzer-Policies müssen mit denen des Cloud-Providers verglichen werden können, um herauszufinden, ob dieser sie überhaupt einhalten kann.

Nach Prüfung der Policy-Sprachen auf Erfüllung obiger Anforderungen folgerte [8], dass keine der Sprachen für den Einsatz in Cloud-Umgebungen geeignet sei. Aus diesem Grunde wurde eine neue Policy-Sprache namens Compact Privacy Policy Language (CPPL) [7] entworfen. Die Besonderheit der Sprache besteht darin, dass die Policies von Menschen in einer leicht verständlichen Form geschrieben werden können, anschließend aber in eine binäre Form umgewandelt werden, wodurch sich der Speicherbedarf reduziert und eine performante Auswertung der Policies ermöglicht wird.

## 2.2 OpenStack

Das SSICLOPS-Projekt hat sich zur Umsetzung seiner Ziele für *OpenStack*[4] als Cloud-Plattform entschieden. OpenStack ist eine Open Source-Sammlung verschiedener Dienste, um eine Cloud nach dem IaaS-Modell aufzubauen. Eine minimale OpenStack-Installation enthält Dienste zur Ausführung von Virtual Machines (VMs) zur Bereitstellung von Rechenkapazität, zum Vernetzen dieser VMs sowie zur Speicherung von Daten. Hinzu kommt ein Dienst zur Nutzerverwaltung. Viele weitere, optionale Dienste unterschiedlicher Reife sind verfügbar [28]. OpenStack ist bei einer Vielzahl von Unternehmen und Forschungseinrichtungen im Einsatz [29]. Dank der Bereitstellung als Open Source-Software bietet sich das Projekt an als Cloud-Plattform für SSICLOPS im Allgemeinen und zur Erweiterung um Unterstützung für Policies im Speziellen.

Zweimal jährlich wird eine neue OpenStack-Version freigegeben. Diese Arbeit baut auf der *Mitaka* genannten Version auf, die im April 2016 veröffentlicht wurde.

## 2.3 Policy-Ansätze in OpenStack

Es gibt in OpenStack bereits Ansätze zur Unterstützung von Policies. Auf einige davon wird im Folgenden eingegangen.

### 2.3.1 oslo.policy

Wie in Abschnitt 2.2 bereits erwähnt, besteht das OpenStack-Projekt aus einer Vielzahl von Diensten. Trotz der klaren Aufgabentrennung zwischen ihnen gibt es Funktionalitäten, die je-

[4] `https://www.openstack.org/`

der Dienst zumindest intern nutzen kann oder gar muss. Das *Oslo*-Projekt[5] erstellt daher eine Reihe von Bibliotheken, die von anderen OpenStack-Diensten genutzt werden können. Einige viel verwendete Bibliotheken vereinfachen und vereinheitlichen zum Beispiel den Zugriff auf Datenbanken und Nachrichtenwarteschlangen, das Caching von Daten, die Konfiguration von OpenStack-Diensten sowie das Logging.

Eine weitere Bibliothek des Oslo-Projekts ist *oslo.policy*. Diese gibt ein Format zur Definition von Regeln und daraus bestehenden Policies vor und stellt darüber hinaus Funktionalitäten zur Durchsetzung der Policies bereit.

Zur Umsetzung von vom Nutzer vorgegebenen Policies, wie im SSICLOPS-Projekt vorgesehen, eignet sich oslo.policy aber nicht. oslo.policy wird zu Zwecken der Autorisierungsprüfung verwendet. Des besseren Verständnisses wegen folgt ein Beispiel für die Anwendung der Bibliothek:

OpenStack-Dienste bieten ihre Funktionalitäten über Application Programming Interfaces (APIs) an, die sowohl durch Menschen als auch andere OpenStack-Dienste genutzt werden können. Der OpenStack-Dienst zur Verwaltung von Nutzern, Gruppen und letztlich der Authentifizierung bietet eine Representational State Transfer (REST)-Schnittstelle zur Änderung von Passwörtern unter dem Pfad `/v3/users/{user_id}/password`.[6] Mittels einer POST-Anfrage lässt sich somit das Password eines Nutzers mit der Nutzer-ID `{user_id}` ändern. Um zu vermeiden, dass jemand mit Kenntnis der Nutzer-ID eines anderen Nutzers dessen Passwort ändert, kommt oslo.policy zur Prüfung der Autorisierung zum Einsatz. Es ermöglicht eine Regel zu konstruieren und durchzusetzen, die den Zugriff auf die API zur Passwortänderung beschränkt. Der Aufruf wird so auf Administratoren und diejenigen Nutzer eingegrenzt, deren Passwort geändert werden soll. Trifft eine Anfrage zur Passwortänderung ein, wird geprüft, ob der anfragende Nutzer dazu berechtigt ist. Der Dienst nutzt dazu oslo.policy und schlägt nach, welche Berechtigung zum Aufruf von `identity:change_password` nötig ist. Dieser Schlüsselwert dient oslo.policy zur Suche in einer dem Dienst beiliegenden Datei, deren Inhalt im JavaScript Object Notation (JSON) [5]-Format vorliegt und für gewöhnlich `policy.json` heißt. Für obigen Schlüsselwert ist in der Standardkonfiguration der Wert `rule:admin_or_owner` eingetragen.[7] Diese Regel wird mithilfe der sich ebenfalls in der Datei befinden Regeldefinition ausgewertet. Der Dienst erfährt dadurch, ob der anfragende Nutzer ein Administrator ist oder sein eigenes Passwort ändern will. Falls ja, ist der Nutzer autorisiert und der Dienst fährt mit der Bearbeitung der Anfrage fort, ansonsten wird für gewöhnlich eine Hypertext Transfer Protocol (HTTP)-Antwort mit dem Statuscode 403 (Forbidden) [3] gesendet.

oslo.policy findet in allen in dieser Arbeit betrachteten OpenStack-Diensten Anwendung und erfüllt die Aufgabe der Autorisierungsprüfung zuverlässig. Der Namensbestandteil *policy* täuscht aber insofern, als dass im SSICLOPS-Kontext Policies sehr viel ausdrucksstärker sein können. Zwar lassen sich auch dort Policies bauen, denen eine Ja/Nein-Antwort genügt

[5] `https://wiki.openstack.org/wiki/Oslo`

[6] `http://developer.openstack.org/api-ref/identity/v3/index.html?expanded=change-password-for-user-detail#change-password-for-user`

[7] `https://github.com/SSICLOPS/policy_keystone/blob/stable/mitaka/etc/policy.json#L49`

(Beispiel: Der Nutzer will eine VM in den USA starten. Darf er das?), es soll aber weitaus mehr möglich sein.

### 2.3.2 Swift Storage Policies

*Swift* ist ein OpenStack-Dienst, der Objektspeicher anbietet. Er ist damit das OpenStack-Äquivalent zu Amazons S3.[8] Die Unterstützung von Policies gehört zu Swifts Kernfunktionen. Objekte werden in einem Container gespeichert. Für jeden Container lässt sich eine Policy anlegen, die beispielsweise eine Replikationsrate für alle sich im Container befindenden Objekte vorschreibt und von Swift umgesetzt wird. Swift Storage Policies können vom Nutzer beim Anlegen eines Containers festgelegt werden, sind allerdings auf Swift beschränkt und für andere OpenStack-Dienste nicht verwendbar.

### 2.3.3 Policy-based Scheduling

Wenn OpenStack eine neue VM starten soll, muss es eine Entscheidung treffen, auf welchem physischen Host sie gestartet werden soll. Dazu werden eine Reihe von Bedingungen an alle zur Verfügung stehenden Hosts gestellt, unter anderem: [26]

- Der Host muss genügend freien RAM haben, um die VM zu starten.
- Falls die VM oder das zu startende Abbild einen bestimmten Hypervisor erfordert, muss dieser auf dem Host verfügbar sein.
- Wenn der Nutzer beim Start der VM explizit eine bestimmte Region vorschreibt (mehr dazu in Abschnitt 7.4), kommen nur die sich darin befindenden Hosts infrage.
- Auch das verbleibende Kontingent des die VM startenden Nutzers muss berücksichtigt werden.

Policy-based Scheduling hatte den Ansatz, die Auswahl eines geeigneten Hosts durch Policies weiter einschränken zu können [16]. Der Betreiber einer OpenStack-Instanz könnte somit eine Policy schreiben, in der er festlegt, dass alle VMs eines bestimmten Kunden nur in einer bestimmten Region gestartet werden dürfen, auch ohne dass der Kunde dies beim Start der VM explizit angeben muss.

Der Nutzer kann die Policy nicht einsehen oder bearbeiten, sondern ist auf eine korrekte Umsetzung seiner Vorgaben durch den Betreiber der Cloud angewiesen. Auf andere OpenStack-Dienste ist Policy-based Scheduling nicht übertragbar.

Dieser Ansatz wurde nicht in die Codebasis von OpenStack übernommen, da der Prozess zum Beitragen von Code nicht vollständig durchlaufen wurde. Details zum Prozess folgen in Kapitel 3.

### 2.3.4 Group Based Policy

Bei Group Based Policy (GBP) handelt es sich um eine Erweiterung für OpenStacks Netzwerk-Dienst *Neutron* [27]. Mittels Policies lässt sich festgelegen, wie mit Netzwerkpaketen zu

[8] `https://aws.amazon.com/s3/`

verfahren ist. Anhand von beispielsweise Port und Protokoll wird ein Paket akzeptiert oder verworfen. Die Prüfung der Policies erfolgt aus Performancegründen nicht für jedes einzelne Paket. Stattdessen werden die Policies in eine virtuelle Netzwerkeinrichtung übersetzt. Dies beinhaltet die Zusammenstellung und Konfiguration virtueller Netze, Switches, Router und Firewalls in einer Art und Weise, dass die Anforderungen der Policies umgesetzt werden. GBP erweitert die Neutron-API, wodurch die Vorgabe von Policies durch den Nutzer ermöglicht wird.

### 2.3.5 Congress

Swift Storage Policies, Policy-based Scheduling und GBP haben gemein, dass sie in der Lage sind, vorgegebene Policies durchzusetzen, allerdings nur in der spezifischen Domäne, für die sie gedacht sind. Das *Congress*-Projekt ist ein separater Service in OpenStack und verfolgt das Ziel, als universeller Anlaufpunkt Compliance in Cloud-Umgebungen zu ermöglichen. Alle bisherigen Policy-Ansätze sollen sich demnach auch mithilfe von Congress umsetzen lassen [30, 35].

Um einen Policy-Verstoß feststellen zu können, benötigt Congress zuallererst Daten, die ausgewertet werden müssen. Dazu fragt Congress regelmäßig die APIs der anderen OpenStack-Dienste ab. Entspricht der dadurch ermittelte Zustand der OpenStack-Instanz nicht den Vorgaben der Policy, so wird der Verstoß protokolliert und der Betreiber der Cloud darüber benachrichtigt. Congress ermöglicht so ein Monitoring hinsichtlich der Einhaltung von Policies.

Die über das bloße Monitoring weit hinausgehende Durchsetzung von Policies gestaltet sich schwieriger. Policies können angeben, wie auf einen festgestellten Verstoß reagiert werden soll. Neben dem bereits vorgestellten Monitoring kann Congress auch versuchen, die Verstöße rückgängig zu machen. Dieser Ansatz ist kaum geeignet, wenn die den Verstoß auslösende Aktion viele Nebenwirkungen hatte, die dann ebenfalls zurückgerollt werden müssen.

Wünschenswert wäre hingegen, wie es die bereits vorgestellten, auf eine Domäne zugeschnitten Ansätze handhaben, einen Verstoß gar nicht erst zuzulassen, sondern ihn proaktiv zu unterbinden. Dazu müssten allerdings die OpenStack-Dienste vor jeder Aktion, die von einer Policy betroffen sein könnte, zuerst Congress konsultieren um herauszufinden, ob sie die Aktion zulassen dürfen oder nicht. Darin liegt ein fundamentales Problem, das auch den in dieser Arbeit vorgestellten Ansatz zur Unterstützung von Policies in OpenStack maßgeblich beeinflusst hat: Jeder einzelne OpenStack-Dienst muss angepasst werden, um Kontakt mit Congress bei relevanten Aktionen aufzunehmen. Da die einzelnen Dienste aber möglichst unabhängig voneinander sind und auch separat entwickelt werden, stellt diese Integration von Congress ein großes Hindernis dar. Zwar unterstützt Congress bereits die Möglichkeit, von anderen Diensten um Erlaubnis gefragt zu werden, der Großteil der bislang von Congress betrachteten Anwendungsfälle bezieht sich jedoch auf das Monitoring [22].

Als Policy-Sprache setzt Congress auf das deklarative *Datalog*, eine Untermenge von *Prolog*. Als Dienst bietet Congress eine API an, die zwar für Betreiber einer OpenStack-Instanz gedacht ist, sich aber mittels oslo.policy (siehe Abschnitt 2.3.1) auch für Nutzer verfügbar machen lässt.

Da bei der Vielzahl von OpenStack-Projekten nicht sofort ersichtlich wird, welche ausgereift genug für einen Produktiveinsatz sind, nimmt die OpenStack Foundation eine Überprüfung der Projekte vor und bewertet anhand verschiedener Kriterien ihre Reife. Congress erreicht dabei im Mitaka-Release vom April 2016 nach einer Entwicklungszeit von zwei Jahren auf einer Skala von 1 bis 8 den Wert 1 [21]. Bei einer im Rahmen dieser Bewertung durchgeführten Umfrage gab 1% der Befragten an, Congress produktiv einzusetzen.

Zwar scheint sich mit Congress ein Weg abzuzeichnen, wie Policies in OpenStack integriert werden, dennoch basiert diese Arbeit nicht auf Congress. Gründe dafür sind:

- Die fehlende Reife und die damit verbundene fehlende Stabilität der APIs. Das Risiko wäre hoch, dass bereits in der nächsten OpenStack-Version Inkompatibilitäten mit den in dieser Arbeit entwickelten Lösungen auftreten und damit den Fortschritt des SSICLOPS-Projekts behindern.
- Die tiefe Integration von Datalog als Policy-Sprache in Congress. Das SSICLOPS-Projekt ist daran interessiert, die Policy-Sprache CPPL (siehe Abschnitt 2.1.2) zu evaluieren.
- Der zeitliche Rahmen dieser Arbeit lässt die Entwicklung eines Forschungsprototypen zu, eine Integration in Congress samt dessen Anpassung zur Nutzung von CPPL erscheint jedoch zu umfangreich.

# 3 Grundlegende Entwurfsentscheidungen

Bevor das Konzept und die Implementierung der Integration von Policies in OpenStack detailliert erläutert werden, wird zunächst auf einige grundlegende Probleme und die daraus jeweils gezogenen Konsequenzen eingegangen.

**Monitoring vs. Durchsetzung von Policies** Wie am Beispiel von Congress (siehe Abschnitt 2.3.5) gezeigt, ist die Durchsetzung von Policies, also die Vermeidung von Verstößen, bevor diese überhaupt eintreten, deutlich schwieriger umzusetzen als reines Monitoring, welches die Verstöße lediglich dokumentiert, sie aber nicht verhindern kann. Wünschenswert wäre die proaktive Unterbindung von Verstößen – Monitoring wirkt deshalb wie ein Kompromiss aus dem Wunsch zur Unterstützung von Policies und technisch einfacher Umsetzbarkeit. Diese Arbeit hat sich zum Ziel gesetzt, Verstöße vor ihrem Auftreten zu unterbinden.

**Vielseitigkeit der Policies** Die Policy-Sprache CPPL (siehe Abschnitt 2.1.2) ist zwar explizit für die Nutzung in Cloud-Umgebungen ausgelegt, macht aber darüber hinaus keine weiteren Angaben, welche spezifischen Policies umsetzbar sein sollen. Der Namensbestandteil *Privacy* in CPPL zeigt zwar die Motivation hinter der Sprache, allerdings finden sich in [7] auch Beispiele, bei denen mithilfe von CPPL nichtdatenschutzspezifische Eigenschaften einer Cloud-Umgebung definiert werden, beispielsweise die Mindestverfügbarkeit von Speichersystemen.

Durch diese Offenheit von CPPL für beliebige Anwendungsfälle lassen sich Policies konstruieren, die jeden OpenStack-Dienst betreffen. Dem Modell von Congress folgend müsste daher jeder Dienst bei jeder Aktion, die von einer Policy betroffen sein könnte, zuerst bei einer zentralen Stelle zur Verwaltung der Policies die Erlaubnis für jene Aktion einholen. Um dies zu erreichen, müssten alle OpenStack-Dienste um Unterstützung für Policies ergänzt werden.

Bei alternativen Ansätzen wie Swift Storage Policies (Abschnitt 2.3.2) oder GBP (Abschnitt 2.3.4), bei denen eine Policy vom betroffenen Dienst selbst durchgesetzt wird, ohne zuerst einen weiteren Service zu kontaktieren, liegt dasselbe Problem vor: Auch hier ist eine Anpassung aller Dienste notwendig. Diese Anpassung ist mit hohem Implementierungsaufwand verbunden, der sich jedoch nicht vermeiden lässt.

**Hoher Implementierungsaufwand** OpenStack-Dienste werden weitestgehend unabhängig voneinander entwickelt. Als Schnittstelle zwischen den Diensten gibt es die dokumentierten APIs. Zusätzlich finden zweimal jährlich im Rahmen der *OpenStack Summits* Treffen der Entwicklergemeinschaft statt, um über Anforderungen und Implementierungspläne für die

folgenden sechs Monate zu diskutieren [23]. Diese Treffen sind geeignet, um projektübergreifende Implementierungen, wie die Integration von Policies, zu planen. Allein diese logistische Komponente ist bereits mit zeitlichem Aufwand verbunden.

**Entwicklungsprozess in OpenStack** Der Quellcode von OpenStack steht auf GitHub zur Verfügung.[9] Zum Beitragen von Code gibt es bei vielen auf GitHub vorhandenen Open Source-Projekten den Prozess, das Repository zu gabeln, Änderungen vorzunehmen und diese mittels eines *Pull Requests* zur Übernahme ins Original-Repository vorzuschlagen. Dieser Vorgang findet bei OpenStack keine Anwendung. Die Quellen auf GitHub sind lediglich gespiegelt; OpenStack betreibt einen eigenen Git-Server[10] mit über 1600 Repositories im Januar 2017.

Die Anforderungen zum Beitragen von Code sind bei OpenStack folgendermaßen: [24, 20]

- Voraussetzung ist ein Account bei der Plattform Launchpad[11].
- Nötig ist ebenfalls eine kostenlose Mitgliedschaft bei der OpenStack Foundation.
- Beitragende müssen einer Lizenzvereinbarung zustimmen.
- Das zu implementierende Feature muss zuerst anhand einer Spezifikation diskutiert werden. Einige Projekte haben zum Verwalten der Spezifikationen und ihrer Revisionen eigene Repositories, andere nicht.
- Es wird auf Launchpad ein sogenannter *Blueprint* erstellt, der auf die Spezifikation verweist. Der Blueprint dient dazu, den Überblick über den Fortschritt bei der Umsetzung des Features zu behalten.
- Das Feature wird in einem eigenen Entwicklungszweig implementiert.
- Sämtlicher Code wird vor der Übernahme in den Hauptzweig umfassend getestet. Dazu betreibt das OpenStack-Projekt eine Instanz des Überprüfungssystems *Gerrit*.[12] Alle Änderungen werden dort mittels automatisierter Tests geprüft. Zusätzlich müssen die Änderungen durch Menschen genehmigt werden.

Dieser Prozess ist mit vergleichsweise viel Arbeit verbunden. Da für die Policies eine Vielzahl von Diensten angepasst werden müssen, bedeutet dies neben der eigentlichen Implementierung einen hohen Aufwand. Ein zentrales Anliegen dieser Arbeit ist es deshalb, den Aufwand für die einzelnen Dienste, die um Policy-Unterstützung erweitert werden sollen, so gering wie möglich zu halten. Dieses Ziel hat diverse Auswirkung auf den Entwurf der umgesetzten Lösung und wird immer wieder aufgegriffen werden.

**Produktivlösung kontra Forschungsprototyp** Wünschenswert wäre, die in dieser Arbeit erstellte Lösung mittels des oben vorgestellten Verfahrens zum Beitragen von Quellcode zur Diskussion zu stellen. Ziel sollte dabei die Integration der Lösung in den offiziellen OpenStack-Code sein. Umsetzbar ist das im zeitlichen Rahmen dieser Arbeit nicht. Aus diesem Grunde

[9] `https://github.com/openstack/`
[10] `https://git.openstack.org/cgit/`
[11] `https://launchpad.net/`
[12] `https://review.openstack.org/`

erfolgte die Einschränkung, einen Forschungsprototypen zu entwickeln, der dem SSICLOPS-Projekt bei der weiteren Arbeit an Policies in OpenStack behilflich ist. Auch diese Entscheidung hat diverse Auswirkungen auf den Entwurf der Lösung.

**Unterstützte Policy-Sprachen** Mit CPPL (Abschnitt 2.1.2) hat das SSICLOPS-Projekt eine Policy-Sprache, die für den Cloud-Einsatz entworfen wurde und nun getestet werden soll. Zur Zeit der Implementierung der in dieser Arbeit vorgestellten Lösung gab es keine in der Programmiersprache von OpenStack verfügbare Implementierung von CPPL. Da der Fokus dieser Arbeit auf der Integration von Policies im Allgemeinen, nicht jedoch von CPPL im Speziellen liegt, wurde eine Lösung entworfen, die unabhängig von der letztlich gewählten Policy-Sprache und deren Notation ist.

Der Einfachheit halber wurde während der Entwicklung mit JSON gearbeitet, auch die im Laufe dieser Arbeit gezeigten Policy-Beispiele werden als JSON ausgedrückt. Welche Schritte notwendig sind, um Unterstützung für CPPL hinzuzufügen, wird in Abschnitt 7.6 erläutert.

# 4 Speicherung von Policies in Keystone

Nutzer kommunizieren mit OpenStack-Diensten über deren APIs. Auch die Kommandozeilenclients der jeweiligen Dienste abstrahieren letztlich nur die offizielle API. Da potentiell jede Anfrage eines Nutzers an die Cloud von einer Policy betroffen sein kann, wäre es hilfreich, wenn die Policy ein Teil der Anfrage an die Cloud wäre. Dieser Ansatz kommt den hinter CPPL stehenden Überlegungen entgegen [7].

Während der Nutzer bei direktem Zugriff auf die API seine Policy leicht selbst mitschicken könnte, würde dieser Ansatz erheblichen Aufwand erfordern, da alle Kommandozeilenclients angepasst werden müssten, um mit Policies umgehen und diese an die API durchreichen zu können. Gleiches gilt für OpenStacks webbasiertes Dashboard. Dieses nutzt zur Kommunikation mit den anderen Diensten deren Kommandozeilenclients, aber auch das Dashboard selbst müsste ebenfalls erweitert werden, um die Policy an die Clients weiterzugeben. Das Dashboard gibt sich beim Kontakt mit anderen Diensten als der am Dashboard angemeldete Nutzer aus – eine über das Dashboard gestartete VM gehört also nicht etwa dem Dashboard selbst, sondern dessen Nutzer.

Wie impraktikabel der Ansatz, den Nutzer die Policy mit jeder Anfrage mitschicken zu lassen, letztlich ist, wird deutlich, wenn man die Software Development Kits (SDKs) bedenkt, die zu vielen OpenStack-Diensten existieren. Damit durch den Nutzer erstellte Software müsste ebenfalls erst angepasst werden, bevor sie Policies durchleiten kann. Wenn nicht nur an OpenStack eine Vielzahl von Änderungen nötig ist, sondern auch die Nutzer noch ihre Tools selbst anpassen müssen, droht das Konzept der Policies aufgrund des Aufwandes und der daher geringeren Nutzung dieses Features zu scheitern.

Wenn der Nutzer die Policy nicht mit jeder Anfrage mitschickt, muss diese in der Cloud selbst gespeichert werden, um sie dort berücksichtigen zu können. Es stellt sich daher die Frage, wo genau die Policy gespeichert werden soll.

Eine Möglichkeit wäre, einen separaten Dienst zu entwerfen, der sich um Policies kümmert, sodass es sich anbietet, diese auch dort zu speichern. Diesen Weg verfolgt Congress. Weshalb diese Arbeit nicht auf Congress aufsetzt, wurde in Abschnitt 2.3.5 erläutert. Einen eigenen Dienst zur Umsetzung von Policies zu erstellen, quasi als Ersatz für Congress, erscheint unnötig viel Arbeit zu sein. Wenn man diesem Weg nicht folgt, muss die Policy in einem bestehenden Dienst gespeichert werden.

Der OpenStack-Dienst *Keystone* ist zuständig für die Authentifizierung von Nutzern. Dazu speichert er bereits Nutzernamen, Passwörter und weitere Details. Da eine Policy zu einem Nutzer gehört, bietet sich Keystone an, um dort ebenfalls Policies zu hinterlegen.

Die folgenden Abschnitte betrachten die Einrichtung einer Entwicklungsumgebung für Keystone und diskutieren, welcher Weg am besten geeignet ist, um Keystone um Funktionen zum Abfragen und Setzen einer Policy zu erweitern.

## 4.1 Entwicklungsumgebung für Keystone

Dieser Abschnitt dokumentiert die notwendigen Schritte, um eine Entwicklungsumgebung für Keystone aufzusetzen. Dadurch soll der Einstieg in eine eventuelle Weiterentwicklung dieser Arbeit vereinfacht werden.

Die OpenStack-Dokumentation stellt eine kurze Anleitung bereit, um die grundlegenden Abhängigkeiten, die für jedes Teilprojekt benötigt werden, zu installieren.[13] Diese Anleitung beschreibt auch den Umgang mit *tox*, dem Tool zum Ausführen von Tests. Empfohlen wird des Weiteren, für jeden Dienst eine separate Umgebung für dessen Abhängigkeiten einzurichten. Da OpenStack eine in der Programmiersprache Python geschriebene Software ist, erfolgt diese Einrichtung mit *virtualenv*.

Die Keystone-spezifische Anleitung zum Einrichten einer Entwicklungsumgebung[14] ist äußerst kurz gehalten. Das Hinzuziehen der Installationsanleitung[15] und der bewährten Praktiken[16] ist hilfreich, aber dennoch nicht ausreichend. Die nötigen Schritte werden daher im Folgenden aufgelistet:

- Voraussetzung ist ein Linux-System; im Kontext von OpenStack sind Ubuntu, Fedora und RHEL/CentOS zu empfehlen, in dieser Reihenfolge [25].
- Zunächst wird der Quellcode mittels *git* heruntergeladen:

  ```
  $ git clone git@github.com:SSICLOPS/policy_keystone.git
  ```

- Alle weiteren Anweisungen geschehen vom geladenen Verzeichnis aus:

  ```
  $ cd policy_keystone
  ```

- Anschließend muss ein *virtualenv* angelegt und aktiviert werden.
- Es folgt die Installation der Abhängigkeiten:

  ```
  $ pip install .
  ```

  Zusätzlich zu den Abhängigkeiten installiert dieser Schritt eine Kopie von Keystone selbst ins virtualenv und erstellt einige Verknüpfungen zum Ausführen und Administrieren von Keystone. Diese Verknüpfungen beziehen sich allerdings auf die Kopie von Keystone, Anpassungen im heruntergeladenen Code werden beim Starten von Keystone mittels der Verknüpfung also nicht ausgeführt. Es hat sich als praktisch erwiesen, die für diese Arbeit relevanten Verknüpfungen aus dem virtualenv ins eigene Arbeitsverzeichnis zu kopieren. Dadurch muss dieser Schritt nicht nach jeder Änderung am Quellcode erneut ausgeführt werden, nur um die Kopie von Keystone im virtualenv aktuell zu halten.

  ```
  $ cp `which keystone-manage` .
  $ cp `which keystone-wsgi-admin` .
  ```

  Die Verknüpfungen beziehen sich dann auf den aktuellen Code.

[13] http://docs.openstack.org/project-team-guide/project-setup/python.html
[14] http://docs.openstack.org/developer/keystone/devref/development.environment.html
[15] http://docs.openstack.org/developer/keystone/installing.html
[16] http://docs.openstack.org/developer/keystone/devref/development_best_practices.html

- Keystone benötigt eine Konfigurationsdatei, die zunächst erstellt und angepasst werden muss.

  Der Einfachheit halber ist bereits eine fertige Konfigurationsdatei enthalten. Um sie einzusetzen, genügt ihre Umbenennung:

  ```
  $ mv etc/keystone.conf.masterthesis etc/keystone.conf
  ```

  Zu Dokumentationszwecken folgt dennoch die Erklärung, wie sie erstellt wurde.

  Keystone liegt bereits eine Beispiel-Konfigurationsdatei bei, die als Basis für die tatsächliche Konfiguration dient:

  ```
  $ cp etc/keystone.conf.sample etc/keystone.conf
  ```

  Die erstellte Datei muss außerdem angepasst werden. Um die Datenbanknutzung zu konfigurieren, ist im `[database]`-Abschnitt der Wert von `connection` auf `sqlite:///keystone.db` sowie im `[identity]`-Abschnitt der Wert von `driver` auf `sql` zu ändern.

  Keystone bietet seine Dienste normalerweise auf zwei Ports an – einer ist zur Nutzung für Administratoren vorgesehen, der andere zum öffentlichen Gebrauch. Die später in dieser Arbeit vorgestellten Dienste begnügen sich mit einem der beiden Ports, das in Kapitel 8 vorgestellte Web-Dashboard benötigt allerdings beide Endpunkte. Die Konfiguration des Dashboards braucht nur einen Port von Keystone zu kennen, den anderen fragt es selbst von Keystone ab. Die Verfügbarkeit auf zwei Ports lässt sich zwar bewerkstelligen, indem Keystone doppelt gestartet wird. Simpler wäre für eine Entwicklungsumgebung allerdings nur die einfache Ausführung. Das lässt sich erreichen, indem Keystone die Anfrage des Dashboards nach dem zweiten Port schlicht mit dem bereits bekannten Port beantwortet. Demnach muss Keystones Konfigurationsdatei für beide Ports denselben Wert angeben: Im Abschnitt `[eventlet_server]` wird der Wert von `public_port` auf `35357` gesetzt, was dem Standard des Administrationsports entspricht.

- Die in der Konfiguration angegebene *sqlite*-Datenbank muss angelegt werden:

  ```
  $ ./keystone-manage db_sync
  ```

- Anschließend erfolgt ihre Initialisierung:

  ```
  $ ./keystone-manage bootstrap --bootstrap-password password \
  --bootstrap-project-name demo \
  --bootstrap-admin-url "http://localhost:\$(admin_port)s/v2.0" \
  --bootstrap-internal-url "http://localhost:\$(public_port)s/v2.0/" \
  --bootstrap-public-url "http://localhost:\$(public_port)s/v2.0/" \
  --bootstrap-region-id RegionOne
  ```

  Die Werte für die Ports werden erst zur Laufzeit anhand der Konfiguration gesetzt, in diesem Beispiel daher immer auf 35357.

- Zum Ausführen von Keystone greifen wir auf *uWSGI* zurück. Dieses kann entweder über die zur Linux-Distribution gehörenden Paketquellen oder via pip installiert werden:

  ```
  $ pip install uwsgi
  ```

  Das zwingend benötigte Python-Plugin ist bei Installation via pip schon dabei, bei anderen Installationswegen muss es möglicherweise selbst nachinstalliert werden.

- Anschließend wird Keystone gestartet:

  ```
  $ uwsgi \
  --http-socket 127.0.0.1:35357 \
  --virtualenv $(dirname $(dirname $(which keystone))) \
  --wsgi-file keystone-wsgi-admin \
  --pyargv "--config-file etc/keystone.conf"
  ```

  Wir legen also Host und Port fest, geben das virtualenv mit den benötigten Abhängigkeiten an, legen die eigentliche zu startende Anwendung fest (eine der kopierten Verknüpfungen) und geben die zu nutzende Konfigurationsdatei an.

- Keystone enthält ein Skript zum Einspielen von Beispieldaten. Dazu gehören unter anderem Nutzer für einige OpenStack-Dienste, Nutzerrollen und die Definition von API-Endpunkten, unter denen die Dienste verfügbar sind.

  Das Skript enthielt Fehler, die in der Version dieser Arbeit jedoch behoben wurden. Das Skript nutzt ein Tool, das im Konflikt zu den Abhängigkeiten von Keystone steht. Daher muss ein neues *virtualenv* angelegt und aktiviert werden. In diesem wird zunächst das Tool, der Kommandozeilen-Client von OpenStack, installiert:

  ```
  $ pip install python-openstackclient
  ```

  Damit das Skript Kontakt zu Keystone aufnehmen kann, benötigt es einige Umgebungsvariablen:

  ```
  $ export OS_USERNAME=admin \
  OS_PASSWORD=password \
  OS_TENANT_NAME=demo \
  OS_AUTH_URL=http://localhost:35357/v2.0
  ```

  Anschließend kann das Skript, während Keystone bereits ausgeführt wird, gestartet werden:

  ```
  $ tools/sample_data.sh
  ```

  Danach ist Keystone vorbereitet auf die Interaktion mit weiteren im Verlauf dieser Arbeit vorgestellten OpenStack-Diensten.

## 4.2 Anpassungen am Datenmodell von Keystone

Keystone verwaltet Nutzer und bietet sich daher auch zur Speicherung von Nutzer-Policies an.

Die Nutzer werden in einer Datenbanktabelle namens *user* gespeichert. Diese soll eine neue Spalte mit dem Namen *policy* bekommen. Da Policies, wie im Falle von CPPL, auch binär kodiert sein können, muss ein passender Datentyp gewählt werden. Weil keine Angaben zur maximalen Länge einer Policy bekannt sind, bietet sich der Datentyp *Binary Large Object (BLOB)* an.

Denkbar wäre auch eine textuelle Darstellung der Policy gewesen. Diese hätte dazu beispielsweise Base64-kodiert werden müssen, hätte dadurch aber mehr Speicher benötigt als die reinen Binärdaten.

Zum Zugriff auf die Datenbank nutzt OpenStack Object-Relational Mapping (ORM) mittels *SQLAlchemy*. Dieses wird auch verwendet, um Datenbankmigrationen zu erstellen. Das Hinzufügen der Policy-Spalte zur Nutzertabelle erfolgt daher letztlich durch das manuelle Anlegen einer Migrationsdatei, deren Schemaänderungen durch Ausführen von `$ ./keystone-manage db_sync` auf die Datenbank angewendet werden.

## 4.3 Abfragen und Setzen einer Nutzer-Policy

Keystone hat bereits APIs zum Abfragen von Details eines Nutzers sowie zum Aktualisieren dieser Daten. Es gibt eine Stelle im Code, an der die beim Abfragen ausgegebenen Attribute eines Nutzers definiert werden. Dort könnte man die Policy hinzufügen und sie wäre Teil der Antwort zu den Nutzerdetails. Analog dazu könnte man beim Aktualisieren der Daten auch einen Wert für die Policy mitschicken, der dann übernommen werden würde. Dieser Ansatz ist naheliegend, bringt jedoch Probleme mit sich:

- Da Policies theoretisch jede Operation in OpenStack betreffen können, müssen sie womöglich sehr oft abgefragt werden. Aus diesem Grunde sind CPPL-Policies binär kodiert, da somit viel Speicherplatz eingespart wird und sich dadurch die benötigte Zeit für eine Abfrage über das Netzwerk verringert.

  Selbst wenn lediglich Interesse an der Policy besteht, so müssten alle Details des Nutzers von Keystone gesendet werden. Diese unnötigen Daten laufen dem Ziel von Kompaktheit und Geschwindigkeit zuwider.

- OpenStack hat eine JSON-API. JSON kann zwar mit Unicode umgehen, nicht aber mit Binärdaten [5]. Da Policies im Falle von CPPL binär kodiert sind, müssen diese vor ihrem Versand zunächst beispielsweise mit Base64 kodiert werden. Das ist zwar unproblematisch, verträgt sich aber schlecht mit dem bisher so einfachen Ansatz: Statt nur das Policy-Attribut in die Liste der auszuliefernden Attribute aufzunehmen, müsste die vorhandene, sehr generische API-Implementierung zur Abfrage von Nutzerdetails angepasst werden, um den Sonderfall der Base64-Kodierung einer Policy zu berücksichtigen. Analog müsste die Policy auch beim Aktualisieren von Nutzerdaten eine Sonderbehandlung erhalten, da sie zuerst dekodiert werden müsste.

Aufgrund dieser Probleme fiel die Entscheidung, separate API-Endpunkte zum Abfragen und Setzen einer Policy anzulegen. Dazu wird in Keystone eine neue Route definiert: `/v3/users/{user\_id}/policy`. Wird diese mittels einer GET-Anfrage aufgerufen, wird die Policy für

den Nutzer mit der ID `user_id` zurückgegeben; beim Aufruf mit einer PUT-Anfrage lässt sich die Policy des Nutzers aktualisieren. Analog zu anderen APIs erfolgt die Integration auf mehreren Ebenen:

- Die neue Route wird in den **Router**-Modulen registriert.
- Eine interne Zuordnung ruft abhängig von der Route einen **Controller** auf, der die Anfrage delegiert an ...
- ... ein **Core**-Modul. Auf dieser Ebene erfolgt die Base64-(De)Kodierung, damit die darunter liegende Ebene ...
- ... des **SQL-Backends** nur noch Daten aus der Datenbank auslesen oder in sie schreiben braucht.

Damit bei häufiger Abfrage der Policy eines Nutzers dieser Stapel nicht jedes Mal neu durchlaufen werden muss, wird auf Ebene des Core-Moduls die fertige Base64-Kodierung in einem Cache gespeichert. Der Cache-Eintrag wird erst bei einer Aktualisierung der Policy invalidiert.

Die neuen API-Endpunkte wurden in die v3-Version der Keystone-API integriert. Sie stehen somit in älteren Versionen nicht zu Verfügung.

Mittels zweier Einträge – einer zum Abfragen, einer zum Aktualisieren einer Policy – in `policy.json` wird mittels oslo.policy (siehe Abschnitt 2.3.1) geregelt, wer Zugriff auf die neuen APIs hat. Zur Abfrage sind Administratoren, andere OpenStack-Dienste sowie Nutzer, wenn es sich um ihre eigene Policy handelt, zugelassen. Wichtig ist hier insbesondere die Abfrage durch andere Dienste: Wenn sie die Policy eines Nutzers berücksichtigen sollen, müssen sie diese von Keystone abfragen, weshalb die Berechtigung benötigen.

Zur Aktualisierung hingegen sind nur Administratoren sowie die jeweiligen Nutzer selbst zugelassen. Für andere OpenStack-Dienste gibt es keinen Grund, die Policies von Nutzern zu aktualisieren.

# 5 Kommunikation mit Keystone

In Abschnitt 4.3 wurden neue APIs zu Keystone hinzugefügt, um Policies abzufragen oder um sie zu aktualisieren.

OpenStack-Dienste können diese APIs nutzen, um mit Keystone zu kommunizieren. Da viele der APIs vor ihrer Nutzung jedoch eine Authentifizierung erfordern, müssten die Dienste sich zuerst darum kümmern.

Um diesen Schritt zu erleichtern und die API weiter zu abstrahieren, gibt es einen in Python geschriebenen Client für Keystone. Dieser kann von anderen Diensten zur einfacheren Kommunikation mit Keystone benutzt werden. Aus Sicht der Dienste ist damit keine Kommunikation über HTTP mehr nötig, sondern ein Aufruf von Methoden in der Client-Bibliothek genügt.

Der Ansatz, für einen Dienst auch einen Client bereitzustellen, ist in OpenStack verbreitet und trifft auf alle in dieser Arbeit vorgestellten Dienste zu.

## 5.1 Anpassungen an python-keystoneclient

Der zu Keystone gehörende Client wird in einem Projekt namens *python-keystoneclient* entwickelt. Da Keystone um die Funktionen zur Policy-Verwaltung ergänzt wurde, bietet sich auch eine Erweiterung dieser Client-Bibliothek an.

python-keystoneclient gruppiert Funktionalitäten in sogenannten *Managern*. Alle zu Nutzern gehörenden Operationen finden sich daher in einem *UserManager*. Dieser wurde um die Methoden *get_policy* und *update_policy* erweitert. Sie greifen auf die Keystone-API mittels HTTP-GET- und -PUT-Anfragen zu. Ihr Vorgehen ist weitgehend trivial, erwähnenswert ist jedoch, dass sie die Base64-Kodierung der Policy vor dem die Bibliothek nutzenden Dienst verbergen. Dieser erhält daher die reine Policy, ob sie nun binär ist oder nicht. *get_policy* nimmt demnach eine Base64-Dekodierung vor, *update_policy* eine Base64-Kodierung.

## 5.2 Entwicklungsumgebung für python-keystoneclient

Die Dokumentation der Bibliothek[17] enthält keine Informationen zum Einrichten einer Entwicklungsumgebung. Dies liegt womöglich am vergleichsweise einfachen Vorgehen, das hier dennoch dokumentiert werden soll:

- Zunächst wird der Quellcode mittels *git* heruntergeladen:

  ```
  $ git clone git@github.com:SSICLOPS/policy_python-keystoneclient.git
  ```

[17] `http://docs.openstack.org/developer/python-keystoneclient/`

- Alle weiteren Anweisungen geschehen vom geladenen Verzeichnis aus:

  ```
  $ cd policy_python-keystoneclient
  ```

- Anschließend muss ein *virtualenv* angelegt und aktiviert werden.
- Es folgt die Installation der Abhängigkeiten:

  ```
  $ pip install .
  ```

Da es sich um eine Bibliothek handelt, kann diese nicht direkt ausgeführt werden. Möglich ist aber beispielsweise ein kurzer Test in einer Python-Konsole, ob bei laufendem Keystone das Abfragen einer Policy möglich ist:

```
from keystoneauth1.identity import v3
from keystoneauth1 import session
from keystoneclient.v3 import client
auth = v3.Password(auth_url='http://localhost:35357/v3',
                   username='admin',
                   password='password',
                   project_name='demo',
                   project_domain_id='default',
                   user_domain_id='default')
sess = session.Session(auth=auth)
keystone = client.Client(session=sess)
keystone.users.get_policy('3ba28375ef5e44458cf97634387aa773')
```

Bei der Authentifizierung müssen womöglich Werte angepasst werden, insbesondere bedarf die Nutzer-ID in der letzten Zeile einer Änderung auf einen existierenden Wert.

Relevant ist des Weiteren die Benutzung dieser Version von python-keystoneclient in anderen Projekten. Dies zeigt sich am besten im Kontext eines konkreten Projektes, zum Beispiel in Abschnitt 6.3.4.

# 6 Aufbau und Konzepte einer Middleware zur Bereitstellung von Policies

Im vorigen Kapitel wurde beschrieben, wo Policies gespeichert und wie sie abgerufen beziehungsweise geändert werden können. Zur Integration von Policies in OpenStack gehört des Weiteren ihre Beachtung durch die eigentlichen Dienste. Dazu müssen diese Zugriff auf die Policy haben, was ihnen durch eine API in Keystone ermöglicht wird.

Ein naheliegender Ansatz ist, dass ein Dienst die Policy jedes Mal abruft, wenn er sie benötigt. Eigentlich ist der Dienst aber nur an der Policy selbst interessiert; woher und über welche API er sie erhält, ist für ihn nicht relevant.

In Kapitel 3 wurde erläutert, dass ein wichtiges Anliegen dieser Arbeit darin besteht, die Integration von Policy-Unterstützung in die einzelnen Dienste so einfach wie möglich zu gestalten. Falls jeder Dienst die Abfrage von Policies selbst implementieren und diese an von Policies betroffenen Stellen zuerst abrufen müsste, würde dies zu Codeduplikation zwischen den Diensten führen. Zudem ist die Abfrage einer Policy durch die Base64-Kodierung nicht trivial. Beides läuft dem Ziel der Einfachheit zuwider. Unter dem Gesichtspunkt, dass langfristig die Unterstützung von Policies womöglich in Congress (siehe Abschnitt 2.3.5) integriert wird, wäre es umständlich, alle Dienste abermals anpassen zu müssen, um die Policy nicht von Keystone zu beziehen, sondern von Congress.

Dieses Kapitel stellt eine Middleware, *policymiddleware* genannt, vor, die den Diensten das Abfragen der Policy abnimmt. Wenn ein Dienst eine Policy auswerten will, ist diese dank der policymiddleware bereits abgefragt worden und muss nicht erst beschafft werden.

## 6.1 Middlewares in OpenStack

Das Konzept von Middlewares in OpenStack hängt eng mit Pythons Web Server Gateway Interface (WSGI) zusammen. Vor der ersten Spezifikation von WSGI [9] im Jahr 2003 waren mit Python-Webframeworks erstellte Webapplikationen schlecht kompatibel zu Webservern, die die Webapplikationen ausliefern sollten. Entwickler mussten sich daher auf eine unterstützte Kombination aus Webframework und Webserver festlegen.

Im Java-Umfeld gab es dieses Problem nicht: Solange die Webapplikation, unabhängig vom Webframework, letztlich als *Servlet* zur Verfügung stand, konnte sie von allen Webservern ausgeliefert werden, die die Servlet-Spezifikation unterstützten.

WSGI soll Ähnliches für Python erreichen: Es beschreibt eine Schnittstelle zwischen Webservern und Webapplikationen beziehungsweise Webframeworks zur geringeren Abhängigkeit der Applikationen von den Servern und damit zur besseren Portabilität.

Auf Serverseite sorgt WSGI dafür, dass die Applikation ausgeführt wird. Eine vom Nutzer eingehende Anfrage wird zusammen mit Umgebungsinformationen und einer Rückruffunktion an die Applikation übergeben. Diese bearbeitet die Anfrage und gibt ihre Antwort mittels der Rückruffunktion an den Server zurück, der letztlich dem Nutzer antwortet.

Zwischen Server und Applikation kann sich eine WSGI-Middleware befinden. Diese implementiert sowohl Server- als auch Clientseite der WSGI-Spezifikation. Für den Server sieht die Middleware aus wie die Applikation, für diese erscheint die Middleware wie der Server. Die Anfrage eines Nutzers gelangt daher zuerst zur Middleware, bevor diese sie an die Applikation weitergibt. Deren Antwort wiederum passiert zunächst die Middleware, bevor der Server sie zur Auslieferung erhält. Die Middleware ist deshalb in der Lage, Anfragen, die beispielsweise gewisse Kriterien nicht erfüllen, gar nicht bis zur Applikation durchzureichen, sondern eine Fehlerantwort zu produzieren und diese vom Server ausliefern zu lassen. Des Weiteren kann eine Middleware sowohl die Anfrage manipulieren, beispielsweise gewisse Informationen im Header ergänzen, als auch die Antworten der Applikation beliebig bearbeiten.

Zwischen Server und Applikation lassen sich beliebig viele Middlewares platzieren, die jeweils nur ihre direkten Nachbarn als Server und Applikation wahrnehmen. Damit lassen sich ganze Middleware-Stapel vor einer Applikation aufbauen.

## 6.2 Beispiel: keystonemiddleware

Um den OpenStack-Diensten eine komfortable Nutzer-Authentifizierung zu ermöglichen, können diese eine spezielle Middleware einsetzen.

Nutzer authentifizieren sich bei Keystone und erhalten bei Erfolg ein Token. Dieses muss der Nutzer bei der Kommunikation mit den OpenStack-Diensten mitschicken. Um die Validität des Tokens zu gewährleisten, müssten die Dienste Token an Keystone zur Überprüfung weiterleiten und auch solche Fälle angemessen behandeln, bei denen die Validierung fehlschlägt. Das ist eine Aufgabe, die die Dienste eigentlich nicht interessiert, relevant ist für sie nur das Wissen, wer der Nutzer tatsächlich ist, der sie gerade kontaktiert.

Um den Diensten diese Verantwortung abzunehmen, gibt es das OpenStack-Projekt *keystonemiddleware*. Wenn OpenStack-Dienste keystonemiddleware einsetzen, durchläuft eine Anfrage an den Dienst zunächst diese Middleware, bevor sie den eigentlichen Dienst erreicht. Der Nutzer muss sein Token im HTTP-Header seiner Anfrage mitschicken. keystonemiddleware nimmt das Token und kontaktiert eine Keystone-API zur Validierung des Tokens.

Falls die Validierung fehlschlägt, gibt die Middleware eine Antwort mit dem HTTP-Statuscode 401 (Unauthorized) zurück. In diesem Falle gelangt die Anfrage überhaupt nicht bis zum eigentlichen Dienst.

Ist die Validierung erfolgreich, fügt die Middleware weitere HTTP-Header hinzu. Diese enthalten unter anderem den Namen und die ID des Nutzers. Die Middleware leitet die Anfrage weiter und der OpenStack-Dienst kann aus den Headern entnehmen, welcher Nutzer ihn gerade kontaktiert, ganz ohne sich selbst um das Token oder die Authentifizierung kümmern zu müssen. Beim Bearbeiten der Antwort der Applikation (beziehungsweise der Middleware

eine Stufe tiefer) ist keystonemiddleware zurückhaltender: Falls der HTTP-Statuscode von einer anderen Komponente auf 401 (Unauthorized) gesetzt wurde, fügt die Middleware einen Header mit Informationen hinzu, wo Nutzer sich authentifizieren können, in der Regel also einen Uniform Resource Locator (URL) zu Keystone.

## 6.3 policymiddleware

In Rahmen dieser Arbeit wurde die Middleware *policymiddleware* entwickelt, die ähnlich wie keystonemiddleware vorgeht: Bevor die Anfrage den eigentlichen Dienst erreicht, holt die Middleware die Policy des anfragenden Nutzers und fügt sie dem Header der Anfrage hinzu. Der Dienst kann daher davon ausgehen, bei einer Anfrage die Policy des Nutzers im Header vorzufinden, wenn er die policymiddleware einsetzt.

Als Schlüssel im Header wurde `X_POLICY` gewählt. Um der im Forschungskontext gewünschten Flexibilität Rechnung zu tragen, wird die Policy nur gesetzt, wenn der Header noch keinen Wert für `X_POLICY` enthält. Vorhandene Werte werden demnach nicht überschrieben. Das erlaubt einem Nutzer, unabhängig von der in Keystone hinterlegten Policy, einzelne Anfragen mit anderen Policies zu verschicken. Andererseits stellt dieses Vorgehen eine Schwachstelle dar: Unter der Annahme, dass im Unternehmenskontext Policies zur Cloud-Nutzung vorgeschrieben werden, kann ein Nutzer die Vorschriften mit diesem Mittel leicht umgehen.

Die Middleware kontaktiert Keystone zur Abfrage der Policy über die in Abschnitt 4.3 beschriebene API. Dazu wird die ID eines Nutzers benötigt. policymiddleware bezieht diese aus den durch keystonemiddleware hinzugefügten Headern – es ist daher erforderlich, dass die Nutzeranfrage zuerst keystonemiddleware passiert, bevor policymiddleware tätig wird. policymiddleware nimmt keine Änderungen an der Antwort der Applikation beziehungsweise der nächsten Middleware vor.

### 6.3.1 Middleware kontra Bibliothek

Der gewählte Middleware-Ansatz ist für den jeweiligen OpenStack-Dienst sehr komfortabel, weil er die Policy nicht selbst abzufragen braucht, sondern sie schon Teil der Anfrage ist. Zugleich ist damit ein Nachteil verbunden: Jede Anfrage führt zu einem Abruf der Policy des Nutzers, unabhängig davon, ob der Nutzer überhaupt eine Policy hinterlegt hat oder ob die angefragte Aktion von einer Policy betroffen ist beziehungsweise Policies berücksichtigt. Vermutlich verbraucht daher der Großteil der abgerufenen Policies unnötig Zeit, die wegen des Abrufs über das Netzwerk nicht zu vernachlässigen ist. Zwar muss die Policy nicht jedes Mal neu aus der Datenbank geladen werden, da Keystone diese nach erstmaligem Abruf wie in Abschnitt 4.3 beschrieben in einem Cache vorhält. Auch die Keystone-Sitzung wird, sofern sie noch gültig ist, von der Middleware wiederverwendet, sodass keine erneute Authentifizierung des Dienstes gegenüber Keystone stattfinden muss. Die Netzwerkkommunikation bleibt dennoch notwendig.

Die Benutzung einer Bibliothek anstelle einer Middleware könnte teils helfen: Der OpenStack-Dienst würde die Policy explizit durch die Bibliothek abfragen lassen, wenn er sie tatsächlich benötigt, die vom Nutzer angefragte Aktion also von einer Policy betroffen ist. Dieser Weg würde den unnötigen Netzwerkverkehr reduzieren, die OpenStack-Dienste müssten jedoch um die Nutzung der Bibliothek erweitert werden. Das Problem des Abfragens von Policies von Nutzern, die überhaupt keine Policies hinterlegt haben, bleibt auch mit einer Bibliothek bestehen.

Zwar sind auch bei Nutzung der Middleware Maßnahmen wie das Caching der Policy auf Seite der OpenStack-Dienste denkbar, die dann notwendigen periodischen Updates beziehungsweise der Umgang mit Benachrichtigungen über eine aktualisierte Policy laufen jedoch dem Ziel zuwider, die Integration in den Diensten so einfach wie möglich zu gestalten.

Diese notwendige Einfachheit hat in Kombination mit dem Vorschlag von CPPL, die Policy als Teil der Anfrage mitzuschicken [7], zur Auswahl des Middleware-Ansatzes geführt.

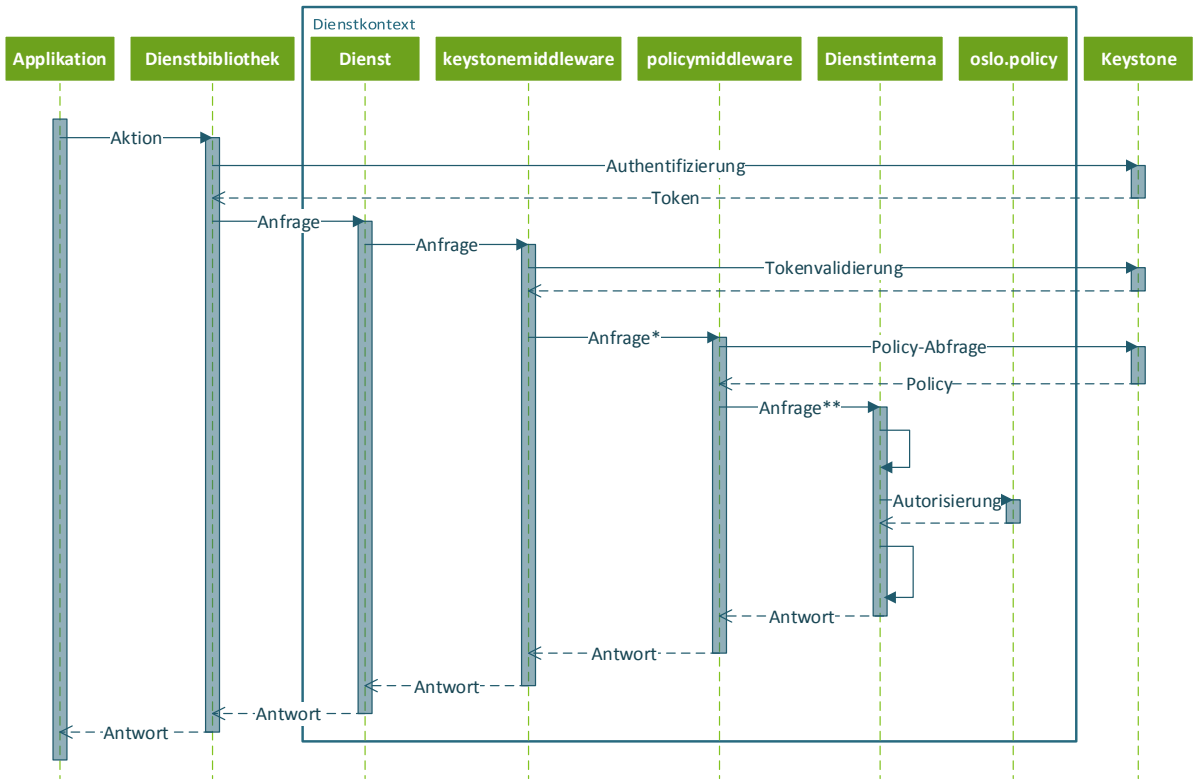


**Abbildung 6.1:** Sequenzdiagramm für die policymiddleware – sie fragt die Policy von Keystone ab und leitet die modifizierte Anfrage an den eigentlichen Dienst weiter.

Abbildung 6.1 zeigt, wie sich Middlewares im Allgemeinen und policymiddleware im Speziellen in den Ablauf einer fehlerfreien Anfrageverarbeitung integrieren. Eine Nutzerapplikation kommuniziert mit den OpenStack-Diensten häufig nicht direkt über deren APIs, sondern nutzt eine Bibliothek des jeweiligen Dienstes. Bevor diese mit dem eigentlichen Dienst Kontakt aufnehmen kann, muss der Nutzer bei Keystone authentifiziert werden. Das bei Erfolg ausgehändigte Token schickt die Bibliothek bei der Anfrage an den Dienst mit. In

dessen Kontext werden zunächst die konfigurierten Middlewares (siehe nächster Abschnitt) durchlaufen. Die keystonemiddleware validiert das Token und modifiziert die Anfrage, indem sie Details zum anfragenden Nutzer in den Header schreibt (*Abfrage**). Auf diese Informationen greift die policymiddleware zu und fragt die Policy des Nutzers von Keystone ab. Die Policy wird dem Header der Anfrage hinzugefügt, wodurch diese abermals modifiziert wird (*Abfrage***). Sie gelangt schließlich zum eigentlichen Dienst und wird dort in dessen Interna verarbeitet. Dazu gehört unter anderem, mittels oslo.policy zu prüfen, ob der Nutzer berechtigt ist, die angefragte Aktion auszuführen. Die Antwort des Dienstes durchläuft die Middlewares abermals, wobei weder keystonemiddleware noch policymiddleware Änderungen an der Antwort durchführen. Letztlich gelangt die Antwort zurück zur Bibliothek, die sie der Applikation ausliefert.

### 6.3.2 Konfiguration

Wie in Abschnitt 4.3 erläutert, darf eine Policy nur von ihrem Nutzer, einem Administrator oder einem OpenStack-Dienst abgefragt werden. Damit policymiddleware Zugriff erhält, muss sie sich daher bei Keystone authentifizieren. Da sie einem Dienst vorgeschaltet ist, soll sie gegenüber Keystone als dieser Dienst auftreten.

Vor der gleichen Herausforderung steht keystonemiddleware bei der Validierung der durch Nutzer gesendeten Tokens, da diese Funktionalität gewisse Berechtigungen voraussetzt. Jeder Dienst hat daher einen eigenen Nutzeraccount bei Keystone. Wenn ein Dienst keystonemiddleware nutzt, muss die Konfigurationsdatei des Dienstes einen Abschnitt enthalten, der der Middleware Nutzername und Passwort des Dienstes zur Authentifizierung bei Keystone mitteilt.

Dieselben Zugangsdaten benötigt auch policymiddleware. Eine Option wäre daher das Auslesen und Benutzen des Konfigurationsabschnittes von keystonemiddleware. Dieser Weg ist fehleranfällig, weil sich Interna der keystonemiddleware ändern können. Aus diesem Grunde wurde auch die Überlegung verworfen, policymiddleware zu einem Teil von keystonemiddleware zu machen. Stattdessen ist policymiddleware eine eigene Middleware und erfordert einen separaten Abschnitt in der Konfigurationsdatei. Neben Nutzername und Passwort wird der URL von Keystone benötigt.

Weil das Auslesen von Konfigurationsdateien eine häufige Aufgabe ist und viele OpenStack-Dienste betrifft, stellt das Oslo-Projekt mit *oslo.config* eine Bibliothek bereit, die sowohl das Auslesen als auch die Definition von erwarteten Parametern beim Programmstart vereinfacht. Die Angabe von Standardwerten für die Parameter ist möglich.

Anhand dieser Erwartungen sowohl eines OpenStack-Dienstes selbst an die Konfigurationsparameter als auch der Erwartungen seiner rekursiven Abhängigkeiten lässt sich automatisch eine Konfigurationsdatei für einen Dienst erstellen. Um die Integration von Policies auch hier so einfach wie möglich zu gestalten, unterstützt policymiddleware die Ermittlung der erwarteten Parameter und ist in autogenerierten Konfigurationsdateien bereits weitestgehend vorkonfiguriert.

Neben dieser Konfigurationsdatei gibt es für jeden Dienst, der WSGI-konform ist (siehe Abschnitt 6.1) eine weitere Datei namens `api-paste.ini`. Sie legt fest, was im WSGI-Stapel

die eigentliche Applikation ist und welche Middlewares zwischen dieser und dem Webserver vorhanden sein sollen. In diese Liste an Middlewares muss auch policymiddleware eingetragen werden – wegen der bereits beschriebenen Abhängigkeit von der Nutzer-ID aber erst nach keystonemiddleware.

### 6.3.3 Integration in oslo.middleware

Manche Anwendungsfälle für Middlewares lassen sich über Projektgrenzen hinweg wiederverwenden. Das Oslo-Projekt entwickelt dafür die Bibliothek *oslo.middleware*. Enthalten ist beispielsweise eine Middleware zum Abfangen sämtlicher Ausnahmen tiefer im Middleware-Stapel oder in der Applikation selbst. Tritt eine Ausnahme auf, gibt die Middleware eine HTTP-Antwort mit dem Statuscode 500 (Internal Server Error) an die nächsthöhere Middleware beziehungsweise den Webserver zurück.

Die Middlewares in oslo.middleware erben alle von derselben Basisklasse und brauchen lediglich, falls gewünscht, die Methoden zum Anpassen von Anfrage und Antwort überschreiben. Dieser Weg macht die Konstruktion weiterer Middlewares komfortabel; auch policymiddleware erbt von jener Basisklasse und wurde deshalb in oslo.middleware integriert. Die Intention von *oslo.middleware*, über Projektgrenzen hinweg wiederverwendbare Middlewares bereitzustellen, passt ebenfalls zu policymiddleware, da für sie die letztliche Applikation nicht nur irrelevant, sondern sogar unbekannt ist. Wichtig ist nur die im vorigen Abschnitt beschriebene Konfiguration im Kontext des jeweiligen Dienstes.

### 6.3.4 Entwicklungsumgebung für oslo.middleware

Die Dokumentation zu oslo.middleware[18] enthält keine Informationen zum Einrichten einer Entwicklungsumgebung. Daher folgt kurz das Vorgehen:

- Zunächst wird der Quellcode mittels *git* heruntergeladen:

  ```
  $ git clone git@github.com:SSICLOPS/policy_oslo.middleware.git
  ```

- Alle weiteren Anweisungen geschehen vom geladenen Verzeichnis aus:

  ```
  $ cd policy_oslo.middleware
  ```

- Anschließend muss ein *virtualenv* angelegt und aktiviert werden.
- Es folgt die Installation der Abhängigkeiten:

  ```
  $ pip install .
  ```

- policymiddleware führt eine zuvor nicht vorhandene Abhängigkeit ein: Zum Zugriff auf die in Keystone gespeicherten Policies wird python-keystoneclient (siehe Abschnitt 5.1) genutzt. Normalerweise verwalten OpenStack-Projekte ihre Abhängigkeiten in einer Datei namens `requirements.txt`. Die dort aufgeführten Pakete werden über Umwege via pip aus Pythons Paketindex PyPI[19] installiert. Die in dieser Arbeit vorgenommenen Anpassungen wurden jedoch nicht auf PyPI veröffentlicht.

[18] `http://docs.openstack.org/developer/oslo.middleware/`
[19] `https://pypi.python.org/pypi`

pip selbst kann mit gewissen Formaten in `requirements.txt` umgehen. So lässt sich beispielsweise angeben, ein Paket statt von PyPI aus einem git-Repository und einem bestimmten Entwicklungszweig zu installieren. Da OpenStack jedoch nicht pip direkt aufruft, sondern die Eigenentwicklung *pbr*[20] dazwischenschaltet, die diese Formate nicht versteht, wurde bislang kein zufriedenstellender Weg gefunden, die angepassten Abhängigkeiten (also python-keystoneclient in diesem Fall) automatisch zu installieren.

Erforderlich ist daher, eine Entwicklungsumgebung für python-keystoneclient wie in Abschnitt 5.2 einzurichten und diese als Abhängigkeit hinzuzufügen:

```
$ pip install -e $PFAD_ZU_PYTHON_KEYSTONECLIENT
```

Die `-e` Option richtet die Abhängigkeit im Entwicklungsmodus ein. Das heißt, dass die Abhängigkeit nicht ins Verzeichnis der Abhängigkeiten kopiert wird, sondern eine Referenz auf das Originalverzeichnis der Abhängigkeit angelegt wird. Das hat den Vorteil, dass Änderungen an der Abhängigkeit sofort für oslo.middleware zur Verfügung stehen.

[20] `http://docs.openstack.org/developer/pbr/`

# 7 Das policyextension-Framework

Die bisherigen Kapitel haben gezeigt, wie Policies gespeichert, abgefragt, aktualisiert sowie mit einer Middleware an jede Anfrage angehängt werden. Stellt ein Nutzer eine Anfrage an einen Dienst, so kommt die Policy des Nutzers ebenfalls beim Dienst an, falls dieser zur Nutzung der Middleware konfiguriert ist.

Eine erfolgreiche Integration von Policies in OpenStack erfordert darüber hinaus eine Interpretation und Berücksichtigung der erhaltenen Policies. Zur Umsetzung dieser Aufgaben wurde das *policyextension*-Framework entworfen, das in diesem Kapitel vorgestellt wird. Dabei wird zunächst auf die Motivation zur Nutzung eines Frameworks eingegangen, anschließend werden beispielhaft drei mithilfe des policyextension-Frameworks umgesetzte Policies erläutert. Abschließend werden ausgewählte Interna des Frameworks beleuchtet sowie die gewählten Verfahren diskutiert.

## 7.1 Motivation

Ein grundlegendes Problem bei der Integration von Unterstützung für Policies besteht im erforderlichen Aufwand: Jeder von Policies betroffene Dienst muss erweitert werden.

Da OpenStack-Dienste APIs anbieten und über diese kommunizieren, passiert eine eingehende Anfrage zunächst die API-Schicht eines Dienstes. Zusammen mit der Anfrage trifft dort die Policy ein; es erscheint also naheliegend, sie dort zu interpretieren und gegen die Policy verstoßende Anfragen abzulehnen. Dieser Ansatz scheitert aus mehreren Gründen:

- Die oberste Schicht der Implementierung der API ist bei allen in dieser Arbeit betrachteten Diensten vor allem zur Validierung der Anfrage und ihrer Parameter verantwortlich. Inhaltlich wird die Anfrage erst in tieferen Schichten der Software bearbeitet. Für die Umsetzung vieler Policies ist die oberste Schicht der API daher ungeeignet.
- Die Policy ließe sich zwar als Parameter an diese tieferen Schichten durchreichen. Sauberes Softwaredesign wäre dies jedoch nicht und würde Anpassungen an vielen Schichten erfordern, was zusätzlichen Aufwand bedeutet.
- Die einzelnen Schichten in OpenStack-Diensten sind klar nach Zuständigkeiten voneinander getrennt. Je nach Policy könnte ihre Umsetzung jedoch in verschiedenen Schichten erfolgen müssen, was dem Konzept der Trennung von Zuständigkeiten widerspricht.
- Je mehr Policies ein Dienst unterstützt, desto mehr Code zur Umsetzung der Policies verteilt sich über die Codebasis. Durch die Vielzahl möglicher Policies könnte das Codeverständnis des gesamten Dienstes leiden: Wenn alle paar Zeilen eine Policy aus-

gelesen und überprüft wird, bläht dies vorhandenen Code auf und macht ihn weniger verständlich.

- Derartiger Policy-Code wäre schwer wartbar.

Wünschenswert wäre hingegen eine Lösung mit folgenden Eigenschaften:

- Interpretation und Umsetzung der Policies an einer zentralen Stelle statt über die Codebasis verteilt
- Minimierung des Aufwandes der Integration, das heißt so wenig Anpassung bestehender Dienste wie möglich
- Wartbarkeit des Codes
- Leichte Erweiterbarkeit des Dienstes um Unterstützung zusätzlicher Policies

Eine Verbesserung ließe sich durch das Auslagern des Codes zur Interpretation und Umsetzung erreichen. An den richtigen Stellen im Code würde lediglich eine Überprüfung aufgerufen werden, in der die übergebene Policy auswertet und daraufhin entsprechend reagiert wird. Den ausgelagerten Code könnte jeder Dienst in einem Modul für die Policy-Umsetzung verwalten. Die Aufrufe dieses Codes würden sich jedoch noch immer über die Codebasis verteilen. Des Weiteren müsste die Policy nach wie vor von der obersten Schicht bis zu den Aufrufen durchgereicht werden.

Alle obigen wünschenswerten Anforderungen werden vom policyextension-Framework erfüllt. Dazu sieht es vor, dass Interpretation und Umsetzung der Policies in sogenannten *PolicyExtensions* erfolgen. Diese haben große Ähnlichkeit zu Plugins. Sie sind nicht Teil der Code-Basis eines Dienstes, die lediglich ein leeres Verzeichnis für die PolicyExtensions enthält. Es ist die Aufgabe des Betreibers einer OpenStack-Cloud, PolicyExtensions zu beziehen. Als Quelle vorstellbar sind Plugin-Verzeichnisse ähnlich zu jenen, die es beispielsweise für das Content-Management-System (CMS) *Wordpress* gibt[21]. Dem Betreiber ist es ebenso möglich, selbst PolicyExtensions zu schreiben.

Der größte Unterschied zu klassischen Plugin-Systemen besteht darin, dass PolicyExtensions keine offiziellen Plugin-Schnittstellen benutzen, sondern selbst angeben, wann ihr Code ausgeführt werden soll. Die einzige für jeden OpenStack-Dienst nötige Anpassung ist daher die Angabe eines Verzeichnisses, aus dem das policyextension-Framework die zu ladenden PolicyExtensions bezieht.

Das Framework schreibt einen gewissen Aufbau für die PolicyExtensions vor. Sie müssen dazu von einer bereitgestellten Basisklasse erben und in einem festgelegten Format angeben, wann ihr Code laufen soll. Diese Angabe referenziert eine Funktion im OpenStack-Code. Das Framework sorgt dafür, dass eine Funktion desselben Namens in der PolicyExtension zuvor ausgeführt wird.

Des besseren Verständnisses wegen folgen nach der Anleitung zur Einrichtung einer Entwicklungsumgebung für das Framework zunächst einige Beispiele, wie eine PolicyExtension aussehen kann.

[21] `https://wordpress.org/plugins/`

## 7.2 Entwicklungsumgebung für policyextension

Die Einrichtung einer Entwicklungsumgebung erfolgt vergleichsweise einfach:

- Quellcode mittels *git* herunterladen:

  ```
  $ git clone git@github.com:SSICLOPS/policy_policyextension.git
  ```

- Ins geladene Verzeichnis wechseln:

  ```
  $ cd policy_policyextension
  ```

- Ein virtualenv anlegen und aktivieren.
- Abhängigkeiten installieren:

  ```
  $ pip install .
  ```

## 7.3 Policy-Beispiel 1: Festplattenverschlüsselung

Die oft mit der Nutzung von Cloud-Ressourcen verbundenen Sorgen lassen sich auf verschiedene Weise adressieren: Auf rechtlichem Wege lässt sich ein umfassender Datenschutz formulieren, an den Cloud-Betreiber gebunden werden können. Policies sind ein weiterer Ansatz, wobei dem Betreiber Vertrauen entgegen gebracht werden muss, dass dieser sich tatsächlich an die vereinbarten Policies hält. Zertifizierungen können weiterhin das Vertrauen in den Betreiber stärken.

Auf technischer Ebene ist Verschlüsselung ein häufiger Ansatz. Falls Daten in einer Cloud gespeichert werden, sorgt die Verschlüsselung durch den Nutzer noch vor der Übertragung dafür, dass der Cloud-Betreiber die Daten nicht einsehen kann. Die großen kommerziellen Anbieter von Cloud-Speichern wie Dropbox[22], Google Drive[23] oder Microsoft OneDrive[24] unterstützen eine solche Verschlüsselung bislang nicht. Daher bieten sich beispielsweise verschlüsselte Container an, die als eine einzige Datei in der Cloud gespeichert, lokal aber mit Software wie TrueCrypt oder dessen Nachfolger VeraCrypt[25] ins Dateisystem eingebunden und bearbeitet werden können. Dieser Ansatz geht mit verringertem Komfort einher, da die Software zur Ver- und Entschlüsselung der Container bedient werden muss.

Eine aus Nutzersicht bequemere Möglichkeit bietet die kommerzielle Software Boxcryptor[26]: Sie unterstützt die Speicherung von Daten unter anderem bei obigen Produkten. Nutzer können ihre Zugangsdaten zu den Anbietern in ihrer Installation von Boxcryptor hinterlegen, welches anschließend die Ver- und Entschlüsselung transparent vornimmt und somit vor dem Nutzer verbirgt. Dieser kann lokal auf unverschlüsselte Daten zugreifen, während sie in der Cloud verschlüsselt gespeichert werden.

[22] https://www.dropbox.com/
[23] https://www.google.com/drive/
[24] https://onedrive.live.com/
[25] https://veracrypt.codeplex.com/
[26] https://www.boxcryptor.com/

Einen ähnlichen Ansatz verfolgt das Projekt *Cloud-RAID* [37]: Nutzer installieren einen Client, der Änderungen an ausgewählten Dateien an einen Server überträgt. Dieser trennt die Datei in mehrere Einzelteile auf und überträgt sie an verschiedene Anbieter von Cloud-Speicher. Zusätzlich zu der ebenfalls stattfindenden Verschlüsselung sind die einzelnen Anbieter somit nur im Besitz von Teilen der Daten.

Für die reine Speicherung von Daten in der Cloud gibt es also mehrere Möglichkeiten, um dem Anbieter die Einsicht zu verwehren. Schwieriger wird es, wenn der Anbieter Operationen auf den Daten durchführen können soll. Mit *homomorpher Verschlüsselung* gibt es dazu einen Ansatz, der jedoch aufgrund geringer Performance noch nicht praxistauglich ist [12].

Wegen der Wichtigkeit des Umgangs mit Daten in der Cloud bietet sich daher die Verschlüsselung von Daten als Beispiel für eine Policy an. In OpenStack können Daten unter anderem auf virtuellen Festplatten gespeichert werden. Der dafür zuständige Dienst heißt *Cinder*. Die Policy soll bewirken können, dass mittels Cinder gespeicherte Daten verschlüsselt werden. Es folgt daher zunächst eine Beschreibung, wie sich eine Entwicklungsumgebung einrichten lässt.

### 7.3.1 Entwicklungsumgebung für Cinder

Die offizielle OpenStack-Dokumentation enthält bereits eine Anleitung zur Einrichtung einer Entwicklungsumgebung[27]. Diese ist jedoch zum einen sehr knapp gehalten, zum anderen führt sie zu einigen Problemen, weshalb im Folgenden die nötigen Schritte erklärt werden:

- Zunächst wird der Quellcode mittels *git* heruntergeladen:

  ```
  $ git clone git@github.com:SSICLOPS/policy_cinder.git
  ```

- Alle weiteren Anweisungen geschehen vom geladenen Verzeichnis aus:

  ```
  $ cd policy_cinder
  ```

- Anschließend muss ein *virtualenv* angelegt und aktiviert werden.

- Es folgt die Installation der Abhängigkeiten:

  ```
  $ pip install .
  ```

  Wie schon bei der Entwicklungsumgebung von Keystone (Abschnitt 4.1) werden dabei auch Verknüpfungen zum Starten von Cinder angelegt. Diese existieren auch hier zunächst nur in der Kopie von Cinder im virtualenv, werden aber des höheren Komforts wegen ins Arbeitsverzeichnis kopiert:

  ```
  $ cp `which cinder-manage` .
  $ cp `which cinder-api` .
  ```

- Cinder benutzt oslo.middleware, wir benötigen jedoch die in Abschnitt 6.3.4 installierte Version:

  ```
  $ pip install -e $PFAD_ZU_OSLO.MIDDLEWARE
  ```

[27] `http://docs.openstack.org/developer/cinder/devref/development.environment.html`

- oslo.middleware wiederum soll die angepasste Version von python-keystoneclient benutzen. Für die nun installierte Version von oslo.middleware wurde zwar ein virtualenv eingerichtet, das jene angepasste Variante von python-keystoneclient verwendet. Für Cinder kommt allerdings ein eigenes virtualenv zum Einsatz, sodass python-keystoneclient separat installiert werden muss:

  ```
  $ pip install -e $PFAD_ZU_PYTHON_KEYSTONECLIENT
  ```

- Damit Cinder das policyextension-Framework nutzen kann, muss es nach Einrichtung wie in Abschnitt 7.2 als Abhängigkeit installiert werden:

  ```
  $ pip install -e $PFAD_ZU_POLICYEXTENSION
  ```

- Cinder wird eigentlich ohne Konfigurationsdatei ausgeliefert. Der Einfachheit halber ist in der für diese Arbeit angepassten Version bereits eine fertige Konfigurationsdatei enthalten. Um sie einzusetzen, genügt ihre Umbenennung:

  ```
  $ mv etc/cinder/cinder.conf.mastersthesis etc/cinder/cinder.conf
  ```

  Zu Dokumentationszwecken folgt dennoch die Erklärung, wie sie erstellt wurde.

  Es ist möglich, anhand aller Konfigurationsoptionen Cinders und der der Abhängigkeiten automatisch eine Konfigurationsdatei zu erstellen. Dabei gibt es mehrere Probleme: Mit

  ```
  $ tox -e genconfig
  ```

  lässt sich zwar die Konfigurationsdatei unter `etc/cinder/cinder.conf.sample` ablegen, die durch das Entfernen des `.sample`-Suffixes zur benutzten Konfiguration wird, allerdings fehlen die Voreinstellungen für policymiddleware. Diese können wie in Abschnitt 6.3.2 beschrieben durchaus automatisiert erfasst werden. Obiger Befehl zur Generierung der Konfiguration liest zum Finden der Abhängigkeiten allerdings die Datei `requirements.txt` aus. Dort wiederum konnten die angepassten OpenStack-Projekte jedoch wie in Abschnitt 4.1 erläutert nicht eingetragen werden, sodass policymiddleware nicht als Abhängigkeit erkannt wird. Aus diesem Grunde müssen folgende Zeilen zur Konfiguration hinzugefügt werden:

  ```
  [policymiddleware]

  identity_uri = http://localhost:35357/v3
  service_user = cinder
  service_password = password
  service_project_name = service
  ```

  Die erste Zeile gibt an, unter welcher URL Keystone erreichbar ist. Dabei wird die zu nutzende API-Version explizit vorgeschrieben, weil policymiddleware keine Verhandlung mit Keystone über die zu nutzende Version unterstützt.

  Des Weiteren muss Cinder zur Validierung von Tokens und damit zur Authentifizierung von Nutzern mit Keystone kommunizieren. Dazu wiederum muss Cinder sich selbst gegenüber Keystone ausweisen können. Der `[keystone_authtoken]`-Abschnitt in der Konfigurationsdatei muss daher um folgende Zeilen ergänzt werden:

```
identity_uri = http://localhost:35357
admin_user=cinder
admin_tenant_name=service
admin_password=password
```

Die erste Zeile gibt abermals an, unter welcher URL Keystone erreichbar ist. Im Gegensatz zur Konfiguration von policymiddleware muss keine spezifische API-Version angegeben werden – diese wird mit Keystone ausgehandelt.

Obige Konfigurationszeilen sind gültig unter der Annahme, dass der `cinder`-Nutzer sich mit dem Passwort `password` authentifizieren kann. Das ist der Fall, wenn Keystone wie in Abschnitt 4.1 beschrieben eingerichtet wurde.

Zur Konfiguration der Datenbanknutzung muss im Abschnitt `[database]` außerdem der Wert für `connection` auf `sqlite:///cinder.db` gesetzt werden.

- Diese *sqlite*-Datenbank muss angelegt werden:

  ```
  $ ./cinder-manage db sync
  ```

- Anschließend ist die Cinder-API bereit zum Start. Um weitere zu Cinder gehörende Dienste zu starten, wäre weitere Einrichtungsarbeit nötig, da die einzelnen Prozesse über eine Nachrichtenwarteschlange miteinander kommunizieren. Für dieses Policy-Beispiel jedoch genügt die Ausführung der API.

  ```
  $ ./cinder-api
  ```

  Die API steht daraufhin auf Port 8776 zur Verfügung.

### 7.3.2 Festplattenverschlüsselung in OpenStack

Mit Cinder angelegte virtuelle Festplatten werden im Folgenden als *Volumes* bezeichnet. Jedes Volume ist von einem bestimmten Typ, dem *VolumeType*.

Cinder unterstützt bereits die Verschlüsselung von Volumes. Deren Ziel unterscheidet sich jedoch von oben vorgestellten Lösungen wie VeraCrypt oder Boxcryptor: Die Volume-Verschlüsselung findet komplett intern von OpenStack statt: Der Nutzer bekommt nicht nur keinen Schlüssel für die Daten ausgeliefert, er bekommt generell nichts von der Verschlüsselung mit. Die Intention dieser Volume-Verschlüsselung ist vor allem der Schutz der gespeicherten Daten vor Diebstahl der physischen Datenträger sowie vor unberechtigtem Zugriff auf die Speicherlösung, die die Volumes bereitstellt [34].

In diesem Beispiel soll eine Policy eine solche Verschlüsselung anfordern, sodass jedes neu zu erstellende Volume verschlüsselt gespeichert wird.

Damit OpenStack überhaupt ein verschlüsseltes Volume erstellen kann, wird zunächst ein VolumeType benötigt, für den Verschlüsselung aktiviert ist:

- Dazu wird als Administrator ein separater VolumeType erstellt. In diesem Beispiel heiße er *LUKS*:

  ```
  $ cinder type-create LUKS
  ```

- Anschließend kann für diesen VolumeType Verschlüsselung aktiviert werden. OpenStack bringt dazu bereits ein Modul zur Verschlüsselung mit, *LuksEncryptor*:

```
$ cinder encryption-type-create \
--cipher aes-xts-plain64 \
--key_size 128 \
--control_location front-end \
LUKS \
nova.volume.encryptors.luks.LuksEncryptor
```

Schlüssellänge und Chiffre sind anpassbar, ebenso der Name des VolumeTypes. Linux Unified Key Setup (LUKS) ist unter Linux ein Standardverfahren zur Verschlüsselung von Festplatten und wird hier von `LuksEncryptor` bereitgestellt.

Anschließend könnte jeder zur Erstellung von Volumes berechtigte Nutzer theoretisch auch verschlüsselte Volumes anlegen. Praktisch würde es jedoch fehlschlagen, da bei Nutzung obiger Entwicklungsumgebung nur die API von Cinder ausgeführt wird, die ebenfalls erforderlichen weiteren Teile von Cinder allerdings nicht. Wären diese eingerichtet und gestartet, würde folgendes Beispiel ein Volume mit einer Größe von 10 GB anlegen:

```
$ cinder create \
--display-name "verschluesseltes Volume" \
--volume-type LUKS \
10
```

Relevant ist nun der Umgang mit Verstößen gegen die Policy, also wie eine Anfrage behandelt wird, die keine Verschlüsselung des Volumes anfordert. Denkbar wäre die implizite Aktivierung der Verschlüsselung – OpenStack verschlüsselt das Volume also von allein. Möglich wäre ebenso, einen Fehler zurückzugeben wegen des Policy-Verstoßes. In diesem Beispiel wurde sich willkürlich für Letzteres entschieden; das Beispiel in Abschnitt 7.5.1 zeigt eine implizite Einhaltung von Policies.

### 7.3.3 Umsetzung der Policy

Um eine Policy umsetzen zu können, muss diese zunächst interpretiert werden. Dazu muss bekannt sein, in welcher Form die Policy vorliegt. Das policyextension-Framework stellt den PolicyExtensions die Policy als Python-Datentyp `dict` (Dictionary) zur Verfügung. Darin können Schlüssel-Wert-Paare gespeichert werden, wobei die Werte selbst zusammengesetzte Datentypen sein können, beispielsweise Listen und `dicts`.

Eine für dieses Policy-Beispiel valide Policy könnte aus Sicht der PolicyExtension wie folgt aussehen:

```
{
    "storage": {
        "encryption": True
    }
}
```

Da Cinder für Festplatten zuständig ist, also für sogenannten *Block Storage*, wurde `storage` als Schlüssel für Vorschriften an Cinder gewählt. Darin wiederum befindet sich ein Wahrheitswert für den Schlüssel `encryption`, der angibt, ob explizit angegebene Verschlüsselung beim Anlegen von Volumes zwingend erforderlich ist.

Das weitere Vorgehen soll anhand des konkreten Quellcodes der PolicyExtension gezeigt werden. Dazu befindet sich (relativ zu Cinders Quellverzeichnis) unter dem Pfad `cinder/policyextensions/volume_encryption_required.py` eine Datei, die die Implementierung der Policy in Form einer PolicyExtension enthält. Sie hat folgenden Inhalt:

```
1  from policyextension import PolicyExtensionBase, PolicyViolation
2
3  from cinder.volume import volume_types
4
5
6  class CinderEncryptedVolumeTypeRequiredExtension(PolicyExtensionBase):
7      func_paths = ['cinder.volume.api.API.create']
8
9      def create(self, func_args, policy):
10         try:
11             if policy['storage']['encryption']:
12                 volume_type = func_args['volume_type'] or volume_types.\
13                     get_default_volume_type()
14                 if not volume_type or not volume_types.is_encrypted(
15                         func_args['context'], volume_type['id']):
16                     msg = "Your policy requires using an encrypted volume type."
17                     raise PolicyViolation(msg)
18         except KeyError:
19             pass
```

Dieser Quellcode wird im Folgenden erläutert.

- In Zeile 1 wird aus dem policyextension-Framework eine Basisklasse für PolicyExtensions sowie die `PolicyViolation`-Klasse für Policy-Verstöße geladen.
- Zeile 3 lädt ein zu Cinder gehörendes Modul, das hilfreiche Funktionalität bezüglich Volumes enthält.
- In Zeile 6 wird die eigentliche PolicyExtension namens `CinderEncryptedVolumeTypeRequiredExtension` definiert, die dazu von der importierten Basisklasse erbt.
- Die in Zeile 7 definierte (in diesem Beispiel nur einelementige) Liste `func_paths` gibt an, vor welcher Originalfunktion aus OpenStack der Code der PolicyExtension laufen soll.

  Das Framework wird bei obigem Wert von `func_paths` in der Extension nach einer Methode namens `create` suchen. Diese wird dann unmittelbar vor ihrem Original ausgeführt, also vor der Methode `create` der Klasse `API` im Modul `api` (beziehungsweise der Datei `api.py`) aus dem zu Cinders Quellverzeichnis relativen Pfad `cinder/volume/`.
- In Zeile 9 wird die Methode `create` definiert, die vor dem darüber referenzierten Original ausgeführt wird.

Das Argument `func_args` ist ein `dict`, in dem alle Werte der an die Originalfunktion übergebenen Argumente, zugeordnet zu ihren dortigen Namen, enthalten sind.

Das Argument `policy` ist die Policy des Nutzers, dessen Anfrage gerade verarbeitet wird. Sie ist wie oben beschrieben ein `dict`.

- Zeile 11 überprüft, ob die Policy überhaupt Verschlüsselung verlangt. Sollte die erhaltene Policy keine Vorschriften für Cinder enthalten, wird durch Zeile 19 nichts unternommen, es kommt also zur Ausführung der Originalfunktion.
- Sollte Verschlüsselung verlangt sein, extrahieren die Zeilen 12 und 13 den vom Nutzer angeforderten VolumeType aus den an die Originalfunktion übergebenen Argumenten. Sollte der Nutzer dazu keine Angaben gemacht haben, wird mithilfe von Cinders `volume_types`-Modul ein eventuell als Standard konfigurierter VolumeType ermittelt.
- In den Zeilen 14 und 15 wird schließlich geprüft, ob die Anfrage des Nutzers zu einer Verletzung der Policy führen würde: Falls weder ein VolumeType vom Nutzer angegeben wurde, noch dafür ein standardmäßig zu benutzender Wert konfiguriert ist, dann wurde demnach kein für Verschlüsselung eingerichteter VolumeType ausgewählt. Dies stellt einen Verstoß gegen die Policy dar. Es handelt sich ebenfalls um einen Verstoß, falls der ermittelte VolumeType nicht für Verschlüsselung eingerichtet ist. Dies lässt sich mithilfe des `volume_types`-Moduls überprüfen.
- Wurde ein Verstoß festgestellt, wird in Zeile 16 eine darauf hinweisende Fehlermeldung erstellt, die anschließend in Zeile 17 als `PolicyViolation` geworfen wird.

  Dadurch kommt es nicht mehr zur Ausführung der Originalfunktion und somit zu keiner Verletzung der Policy – diese wurde zuvor erkannt und unterbunden. Der aufrufende Nutzer erhält stattdessen die Fehlermeldung.

Des Weiteren lassen sich einige Eigenschaften des Aufbaus einer PolicyExtension beobachten:

- PolicyExtensions sind Klassen, die von `PolicyExtensionBase` erben.
- `func_paths` muss zwingend vorhanden sein und gibt die Originalfunktion an. Vor ihr soll die gleichnamige Methode der PolicyExtension ausgeführt werden. Eine solche Methode muss demnach existieren.
- Die Argumentliste dieser Methode ist immer gleich: Neben `self` stehen `func_args` und `policy` zur Verfügung.
- Liegt kein Verstoß gegen die Policy vor, unternimmt die Erweiterung nichts. Die Originalfunktion wird anschließend automatisch ausgeführt.
- Liegt ein Verstoß vor, muss die Erweiterung eine `PolicyViolation` werfen.

Es soll an dieser Stelle hingewiesen werden auf `func_paths` und `func_args`: Es ist kaum möglich, eine funktionierende PolicyExtension zu schreiben, ohne zu wissen, wie die ersten Schichten der API-Implementierung des jeweiligen OpenStack-Dienstes grob funktionieren. Dieses Verständnis ist nötig, um mittels `func_paths` die richtige Stelle zur Ausführung der

Erweiterung festzulegen. Der Zugriff auf `func_args` setzt zudem Wissen über die in der Originalfunktion verfügbaren Argumente voraus.

Eine PolicyExtension lässt sich somit zwar mit wenigen Zeilen Code schreiben, erfordert aber durchaus Recherche im Quellcode des OpenStack-Dienstes.

Nur an einer einzigen Stelle benötigt der originale Quellcode von Cinder eine Anpassung: Beim Start sollen auch die PolicyExtensions geladen werden. Dieser wird vor allem von der Datei `cinder/service.py` koordiniert. Sie wurde um drei Zeilen ergänzt:

```
import policyextension

policy_dir = os.path.join(os.path.dirname(__file__), 'policyextensions')
policyextension.load_from(policy_dir)
```

Dieser Code erstellt den Pfad für das Verzeichnis, das die PolicyExtensions enthält und übergibt ihn an die Funktion `load_from` des policyextension-Frameworks. Diese kümmert sich um den Aufruf der Erweiterungen an den von ihnen in `func_paths` angegebenen Stellen. Abschnitt 7.7 geht detailliert auf Interna des Frameworks ein.

## 7.4 Policy-Beispiel 2: Einschränkung der Availability Zones

Neben der Verschlüsselung von Daten ist ein weiterer Anwendungsfall für Policies die Angabe geographischer Vorgaben für die Speicherung von Daten oder die Ausführung von VMs.

Die Platzierung von Daten und VMs in geographischer Nähe zum Nutzer sorgt einerseits für geringe Latenzen bei Nutzung der Cloud-Ressourcen. Anderseits ist die Auswahl der Standorte der physischen Rechenzentren aus rechtlicher Sicht relevant, um womöglich unerwünschten staatlichen Zugriff zu vermeiden. Wie in Kapitel 1 erläutert, sind sich Anbieter kommerzieller Clouds solcher Anforderungen bewusst und bieten für europäische Kunden zunehmend mehr Rechenzentren in Europa an. Nutzer könnten somit die Anforderung haben, explizit erwünschte Standorte auszuwählen oder aber einzelne Standorte auszuschließen.

Dieser Abschnitt wird zunächst die bereits in OpenStack vorhandenen Konzepte zur Unterteilung einer physischen Cloud-Infrastruktur vorstellen. Anschließend wird die Umsetzung einer Policy, die sich auf eines dieser Konzepte bezieht, erläutert.

### 7.4.1 Segregation physischer Infrastruktur

OpenStack bietet mehrere Möglichkeiten zur Trennung physischer Infrastruktur in verschiedene Teile. Im Folgenden werden vier Konzepte zur Segregation der Rechenressourcen kurz vorgestellt, eine ausführlichere Beschreibung findet sich unter [31].

**Host Aggregates** Physische Hosts lassen sich mittels *Host Aggregates* gruppieren. Zielgruppe dieser Funktionalität sind die Betreiber der Infrastruktur: Sie können verschiedene Hosts mit ähnlichen Eigenschaften, beispielsweise Geschwindigkeit der Netzwerkanbindung oder Vorhandensein einer Grafikkarte, zusammenfassen.

Die Host Aggregates können weiterhin mit Metadaten versehen werden. Wenn ein Nutzer eine VM startet, muss er dabei einen bestimmten Typ, genannt *Flavor*, auswählen. Dieser legt unter anderem die Anzahl der Prozessorkerne und die Größe des Arbeitsspeichers fest. Ein Flavor kann ebenfalls um Metadaten ergänzt werden. Deren Angabe erfolgt durch Schlüssel-Wert-Paare. Wird eine VM gestartet, so inspiziert OpenStack die Metadaten des gewählten Flavors und sucht ein dazu passendes Host Aggregate aus. Die darin enthaltenden Hosts kommen zur Ausführung der VM infrage. Beispielsweise könnte so mittels Metadaten ein auf Benutzung von Solid State Disks (SSDs) zugeschnittener Flavor mit einem Host Aggregate assoziiert werden, dessen Hosts SSDs einsetzen.

**Availability Zones** Das Konzept der *Availability Zones* richtet sich an die Nutzer. Auch hierbei handelt es sich um eine logische Zusammenfassung physischer Hosts. Tatsächlich sind Availability Zones durch Host Aggregates implementiert. Die Gruppierung der Hosts erfolgt bei Availability Zones jedoch beispielsweise anhand der Stromversorgung: Alle Hosts, die über denselben Stromkreislauf gespeist werden, könnten eine Availability Zone bilden; jene an einem anderen Stromkreis bilden eine zweite.

Ziel der Availability Zones ist daher, Verfügbarkeit sicherzustellen, auch wenn eine Availability Zone ausfällt. Endnutzer können daher ihre Applikation explizit in mehreren Availability Zones zur Verfügung stellen und sind so besser gegenüber Ausfällen abgesichert.

**Cells** Während die vorigen beiden Ansätze die Ressourcen innerhalb eines Rechenzentrums strukturieren, so erlauben *Cells* die Verteilung einer Cloud auf geographisch verschiedene Standorte. Die Nutzer sind sich dessen überhaupt nicht bewusst, da sie nach wie vor nur über einen einzigen API-Endpunkt mit den Rechenressourcen kommunizieren. Dieser allerdings delegiert beispielsweise die Anfrage zur Erstellung einer VM an eine der Cells, dort wiederum wird ein passender Host ausgewählt.

Cells dienen somit der Skalierbarkeit der Cloud, da nicht nur vorhandene Cells vergrößert, sondern auch weitere Cells hinzugefügt werden können.

**Regions** Ebenso wie Cells eignen sich *Regions* zum Zusammenschluss einer geographisch verteilten Cloud-Infrastruktur. Das Konzept der Regions allerdings wird den Nutzern zugänglich gemacht. Sie können sich daher für bestimme Regions entscheiden und diese explizit ansprechen. Die Regions sind weitgehend voneinander unabhängige OpenStack-Instanzen mit eigenen API-Endpunkten, die sich lediglich den Authentifizierungsdienst Keystone teilen. Nutzer brauchen sich somit auch weiterhin nur einmal zu authentifizieren und können anschließend auf alle Regions zugreifen, sofern sie dazu autorisiert sind. Verglichen mit Cells ist der Aufwand zur Einrichtung einer neuen Region für den Betreiber einer Cloud größer, weil die Regions auch Infrastruktur zur Kommunikation der einzelnen OpenStack-Dienste untereinander, beispielsweise Nachrichtenwarteschlangen, benötigen und diese erst eingerichtet werden müssen.

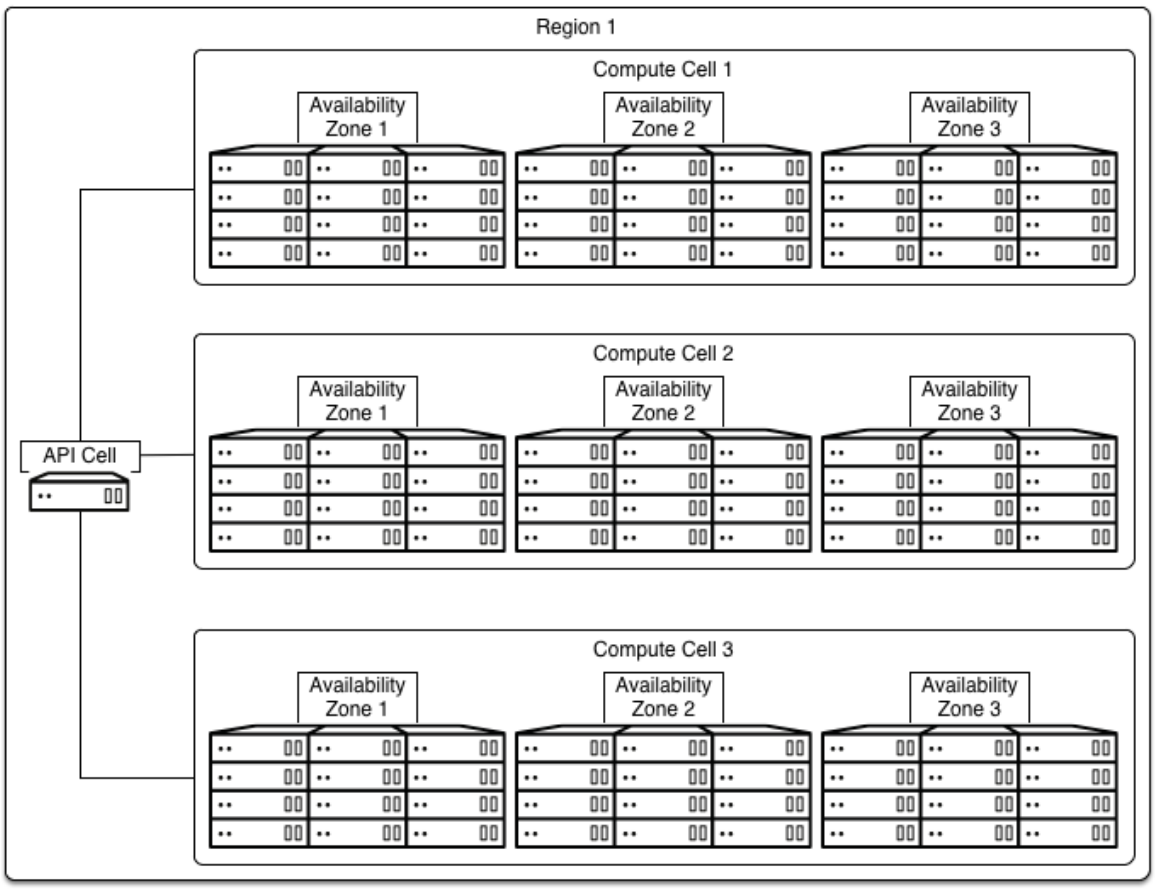


**Abbildung 7.1:** Beispielhafte Strukturierung einer OpenStack-Infrastruktur: Eine Region mit drei Cells, die jeweils drei Availability Zones enthalten. Host Aggregates sind nicht abgebildet, werden jedoch intern zur Implementierung der Availability Zones benutzt [32].

Abbildung 7.1 veranschaulicht die Beziehungen der vorgestellten Konzepte zueinander und ihre Hierarchie. Deutlich wird dabei, dass Cells sich auch innerhalb einer Region nutzen lassen. Konkret könnte das mittels mehrerer zueinander naher Rechenzentren umgesetzt sein.

Im Rahmen des SSICLOPS-Projekts sind Regions das am ehesten geeignete Konzept zur Umsetzung föderierter Clouds. In der Vision einer europäischen Cloud betreiben verschiedene Provider separate OpenStack-Instanzen, die als Regions in eine gemeinsame Cloud eingebettet sind. Nutzer können sich beim zentralen Keystone anmelden und anschließend die Ressourcen aller Anbieter nutzen.

Wünschenswert wäre daher, dass Nutzer mithilfe ihrer Policy ausdrücken können, welche Regions sie ausschließen beziehungsweise auf welche Regions sie sich beschränken wollen. Wie oben bereits erwähnt, ist die Einrichtung von Regions mit nicht unerheblichem administrativen Aufwand verbunden. Dieser Aufwand erschien zum Testen der Umsetzung einer einzelnen Policy nicht gerechtfertigt. Stattdessen wurde der Einfachheit wegen entschieden, die Policy anhand von Availability Zones umzusetzen. Damit können Nutzer noch immer

örtliche Einschränkungen vorgeben; gleichzeitig wird das Testen der Umsetzung erheblich leichter, da nur eine einzelne OpenStack-Instanz benötigt wird oder sogar eine Entwicklungsumgebung (siehe nächster Abschnitt) ausreicht. Diese örtlichen Einschränkungen beziehen sich dann in der Praxis zwar womöglich auf physische Racks oder verschiedene Räume innerhalb eines Rechenzentrums, die Parallelen zu geographisch verteilten Regions erscheinen jedoch ausreichend.

### 7.4.2 Entwicklungsumgebung für Nova

Der OpenStack-Dienst zur Bereitstellung von Rechenressourcen und damit letztlich von VMs heißt *Nova*. Die offizielle OpenStack-Dokumentation enthält eine Anleitung zur Einrichtung einer Entwicklungsumgebung für Nova[28]. Wie schon bei der Entwicklungsumgebung für Cinder (Abschnitt 7.3.1) ist diese jedoch recht rudimentär gehalten, weshalb die nötigen Schritte im Folgenden erklärt werden. Der Ablauf ist teils analog zu Cinder.

- Zunächst wird der Quellcode mittels *git* heruntergeladen:

  ```
  $ git clone git@github.com:SSICLOPS/policy_nova.git
  ```

- Alle weiteren Anweisungen geschehen vom geladenen Verzeichnis aus:

  ```
  $ cd policy_nova
  ```

- Anschließend muss ein *virtualenv* angelegt und aktiviert werden.
- Es folgt die Installation der Abhängigkeiten:

  ```
  $ pip install .
  ```

  Des Komforts wegen werden die Verknüpfungen zum Starten von Nova aus der Kopie im *virtualenv* ins Arbeitsverzeichnis kopiert:

  ```
  $ cp `which nova-manage` .
  $ cp `which nova-api` .
  ```

- Nova benutzt oslo.middleware, wir benötigen jedoch die in Abschnitt 6.3.4 installierte Version:

  ```
  $ pip install -e $PFAD_ZU_OSLO.MIDDLEWARE
  ```

- oslo.middleware wiederum soll die angepasste Version von python-keystoneclient benutzen. Für die nun installierte Version von oslo.middleware wurde zwar ein virtualenv eingerichtet, das die angepasste Version von python-keystoneclient verwendet, allerdings kommt für Nova ein eigenes virtualenv zum Einsatz, sodass python-keystoneclient separat installiert werden muss:

  ```
  $ pip install -e $PFAD_ZU_PYTHON_KEYSTONECLIENT
  ```

- Damit Nova das policyextension-Framework nutzen kann, muss es nach Einrichtung wie in Abschnitt 7.2 als Abhängigkeit installiert werden:

  ```
  $ pip install -e $PFAD_ZU_POLICYEXTENSION
  ```

[28] `http://docs.openstack.org/developer/nova/development.environment.html`

- Nova wird eigentlich ohne Konfigurationsdatei ausgeliefert. Der Einfachheit halber ist in der für diese Arbeit angepassten Version bereits eine fertige Konfigurationsdatei enthalten. Um sie einzusetzen, genügt ihre Umbenennung:

  ```
  $ mv etc/nova/nova.conf.masterthesis etc/nova/nova.conf
  ```

  Zu Dokumentationszwecken folgt dennoch die Erklärung, wie sie erstellt wurde.

  Zwar lässt sich wie schon bei Cinder mit

  ```
  $ tox -e genconfig
  ```

  eine Konfigurationsdatei anlegen, die unter `etc/nova/nova.conf.sample` gespeichert wird. Das anschließende Entfernen des `.sample`-Suffixes führt dazu, dass Nova die Datei nutzt. Allerdings werden auch hier keine vorkonfigurierten Werte für policymiddleware aufgenommen, sodass diese manuell hinzugefügt werden müssen:

  ```
  [policymiddleware]

  identity_uri = http://localhost:35357/v3
  service_user = nova
  service_password = password
  service_project_name = service
  ```

  Des Weiteren muss Nova zur Validierung von Tokens und damit zur Authentifizierung von Nutzern mit Keystone kommunizieren. Damit Nova sich selbst gegenüber Keystone ausweisen kann, muss der `[keystone_authtoken]`-Abschnitt in der Konfigurationsdatei um folgende Zeilen ergänzt werden:

  ```
  identity_uri = http://localhost:35357
  admin_user=nova
  admin_tenant_name=service
  admin_password=password
  ```

  Obige Konfigurationszeilen sind gültig unter der Annahme, dass der `nova`-Nutzer sich mit dem Passwort `password` authentifizieren kann. Das ist der Fall, falls Keystone wie in Abschnitt 4.1 beschrieben eingerichtet wurde.

- Nova braucht zum Starten ein Verzeichnis zur Ablage von Sperrdateien. Dazu findet sich in `etc/nova/nova.conf` im `[oslo_concurrency]`-Abschnitt eine auskommentierte Zeile zum Setzen eines Wertes für den Schlüssel `lock_path`. Diese Zeile muss einkommentiert und mit dem absoluten Pfad eines vorhandenen Verzeichnisses als Wert versehen werden.

- Unter der Annahme, dass die bisherige Einrichtung als regulärer Benutzer, also nicht als `root`, vorgenommen wurde, ist ein weiterer Schritt vonnöten. Nova bringt ein Hilfsprogramm namens `nova-rootwrap` mit, das durch den `root`-Benutzer aufgerufen werden können muss. Dieses befindet sich allerdings innerhalb von Novas Kopie im Verzeichnis von Novas Abhängigkeiten. Dieses Verzeichnis in nicht in der `$PATH`-Variable des `root`-Nutzers enthalten, sodass dieser das Programm nicht findet. Es wird daher eine Verknüpfung in einem Verzeichnis erstellt, das Teil von `$PATH` ist:

  ```
  # ln -s $(which nova-rootwrap) /usr/local/bin/nova-rootwrap
  ```

- Nova sucht beim Start standardmäßig nach diversen Konfigurationsdateien im Pfad `/etc/nova/`. Dieses Verhalten lässt sich zwar über Kommandozeilenparameter und Anpassungen der Konfiguration ändern, am einfachsten ist es jedoch, eine Verknüpfung zum Nova beiliegenden `etc/`-Verzeichnis zu erstellen:

  ```
  # ln -s $(pwd)/etc/nova /etc/nova
  ```

- Eine *sqlite*-Datenbank muss angelegt werden:

  ```
  $ ./nova-manage db sync
  ```

- Anschließend kann die Nova-API gestartet werden. Um weitere zu Nova gehörende Dienste zu starten, wäre zusätzliche Einrichtungsarbeit nötig, da die Prozesse über eine Nachrichtenwarteschlange kommunizieren. Für dieses Policy-Beispiel jedoch genügt die Ausführung der API.

  ```
  $ ./nova-api
  ```

  Die API steht daraufhin auf Port 8774 zur Verfügung.

### 7.4.3 Umsetzung der Policy

Wie schon bei der Festplattenverschlüsselung für Cinder in Abschnitt 7.3.3 soll auch die Umsetzung der Policy zur Einschränkung der Availability Zones erläutert werden.

Grundvoraussetzung ist, dass einem Nutzer die zur Verfügung stehenden Availability Zones bekannt sind. Diese Annahme ist kein Hindernis, da sie über OpenStacks Weboberfläche einsehbar sind und nicht zuletzt mittels einer API abgefragt werden können.

Darüber hinaus ist festzulegen, in welcher Form Nutzer ihre Vorgaben angeben sollen. Insbesondere ist zu entscheiden, ob sie erlaubte Availability Zones vorgeben, also eine weiße Liste pflegen, oder aber Availability Zones ausschließen, also eine schwarze Liste angeben.

Die Entscheidung fiel im Ausschlussverfahren zugunsten der weißen Liste. Im Falle einer schwarzen Liste wäre die Motivation des Nutzers womöglich, gezielt Availability Zones in Staaten mit unzureichendem Datenschutz auszuschließen. Wenn der Cloud-Anbieter eine neue Zone hinzufügt, müsste der Nutzer sofort tätig werden und seine schwarze Liste aktualisieren, ansonsten könnten ungewollt Daten in einem als unsicher erachteten Land verarbeitet werden. Bei einer weißen Liste hingegen kann dies nicht passieren – hier besteht nur der Nachteil, dass eine hinzukommende und für den Nutzer attraktive Zone nicht automatisch zur Verfügung steht, sondern erst explizit genehmigt werden muss. Dieser Ansatz wird aus Gründen des Datenschutzes bevorzugt.

Unter der Annahme, dass eine Availability Zone namens `az2` existiert, könnte eine für dieses Policy-Beispiel valide Policy aus Sicht einer PolicyExtension wie folgt aussehen:

```
{
    "availability_zones": ["az2"]
}
```

Das weitere Vorgehen soll anhand des konkreten Quellcodes der PolicyExtension gezeigt werden. Dazu befindet sich (relativ zu Novas Quellverzeichnis) unter dem Pfad

nova/policyextensions/restrict_availability_zones.py eine Datei, die die Implementierung der Policy in Form einer PolicyExtension enthält. Sie hat folgenden Inhalt:

```
1  import random
2
3  from policyextension import PolicyExtensionBase, PolicyViolation
4
5
6  class AvailabilityZoneRestrictionExtension(PolicyExtensionBase):
7      func_paths = ['nova.compute.api.API.create']
8
9      def create(self, func_args, policy):
10         availability_zone = func_args['availability_zone']
11         try:
12             az_whitelist = policy['availability_zones']
13             if availability_zone:
14                 if availability_zone not in az_whitelist:
15                     msg = ("Your policy does not allow the availability zone "
16                            "you selected.")
17                     raise PolicyViolation(msg)
18             elif az_whitelist:
19                 func_args['availability_zone'] = random.choice(az_whitelist)
20         except KeyError:
21             pass
```

Die grundlegende Struktur des Aufbaus ist durch die Basisklasse `PolicyExtensionBase` vorgegeben und somit identisch mit der Policy zur Festplattenverschlüsselung aus Abschnitt 7.3.3. Im Folgenden wird daher nur kurz auf die wesentlichen Teile der Implementierung eingegangen:

- Die in Zeile 7 angegebene Originalfunktion erhält als optionale Eingabevariable `availability_zone`. Ihr Wert wird in Zeile 10 ausgelesen.
- In Zeile 12 wird die weiße Liste von Availability Zones aus der Nutzer-Policy extrahiert. Falls diese keine Angaben zu Availability Zones enthält, hat die PolicyExtension durch die Zeilen 20 und 21 keine weiteren Auswirkungen auf die Verarbeitung der Anfrage.
- Falls durch den Nutzer eine Availability Zone angegeben wurde, so prüft Zeile 14, ob diese durch die Policy zugelassen ist.
- Ist die gewählte Availability Zone nicht erlaubt, wird in den Zeilen 15 und 16 eine Fehlermeldung formuliert, die in Zeile 17 geworfen wird.
- Falls der Nutzer keine Availability Zone angegeben hat, die Policy diese jedoch einschränkt, so wird in Zeile 19 zufällig eine der zugelassenen Availability Zones ausgewählt.

Zum Laden der PolicyExtensions genügt wie schon bei Cinder die Anpassung bestehenden Quellcodes an nur einer Stelle. Den Großteil der Arbeit beim Starten von Nova übernimmt die Datei `nova/service.py`. Damit die PolicyExtensions beim Start mitgeladen werden, mussten folgende drei Zeilen ergänzt werden:

```
import policyextension

policy_dir = os.path.join(os.path.dirname(__file__), 'policyextensions')
policyextension.load_from(policy_dir)
```

Wie bei Cinder wird hier der Pfad zum Verzeichnis, das die PolicyExtensions enthält, zusammengesetzt und dieser an die Funktion `load_from` des policyextension-Frameworks übergeben, das sich um alles Weitere kümmert.

## 7.5 Policy-Beispiel 3: Replikation

Mit der Nutzung von Cloud-Ressourcen ist ein gewisser Kontrollverlust verbunden. Um sich gegen Ausfälle oder den Verlust von Daten zu wappnen, kann die Replikation von Daten ein Ansatz sein. Aus diesem Grunde verteilt [37] die in der Cloud zu speichernden Daten auf mehrere Anbieter und erzeugt zusätzliche Wiederherstellungsdaten, um den Ausfall einzelner Provider verkraften zu können.

Da Nutzer ein Interesse am Erhalt ihrer Daten haben, ist Replikation in der Cloud wünschenswert. Aus diesem Grunde ist die dritte in dieser Arbeit vorgestellte Policy im Bereich der Replikation angesiedelt.

Denkbar ist, dass ein Nutzer eine minimale Replikationsrate voraussetzt, bevor die Ressourcen eines Anbieters für ihn infrage kommen. Diese Anforderung soll durch eine Policy auszudrücken sein. Die folgenden Abschnitte diskutieren, worauf sich die Replikation beziehen soll, ob OpenStack zuständig für repliziert ausgeführte Anwendungen ist und stellt schließlich die Implementierung der Policy als Ergebnis der Diskussionen vor.

### 7.5.1 Replikation von Anwendungen oder von VMs?

andere Überschrift?

Für das SSICLOPS-Projekt ist es neben der eigentlichen Unterstützung von Policies in Cloud-Umgebungen außerdem relevant zu zeigen, wie in der Cloud ausgeführte Applikationen davon profitieren können.

Als Applikation wurde für SSICLOPS die In-Memory-Datenbank *Hyrise* [14] ausgewählt. Für Hyrise wurde ein Replikationssystem namens *Hyrise-R* [38] entwickelt. Dieses sieht vor, wie aus Abbildung 7.2 ersichtlich, neben einer einzelnen Master-Instanz von Hyrise optional eine oder mehrere Repliken zu betreiben. Der Master steht für Lese- und Schreiboperationen zur Verfügung, während auf die Repliken nur lesend zugegriffen werden kann. Zusätzlich existiert ein Dispatcher, der Anfragen auf die Knoten verteilt. Dazu müssen sich sowohl der Master als auch die Repliken beim Dispatcher

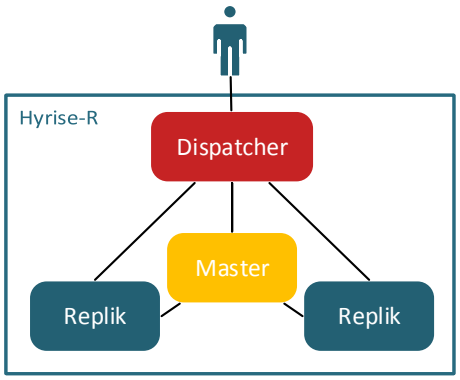


**Abbildung 7.2:** Grundlegende Architektur von Hyrise-R in einem Beispiel mit zwei Repliken

registrieren. Um Änderungen durch erfolgte Schreibzugriffe an die Repliken weiterreichen zu können, müssen sich diese außerdem beim Master registrieren.

Dieser Aufbau kann dank der Verarbeitung jener Anfragen mit ausschließlich lesendem Zugriff durch die Repliken zu einer Leistungssteigerung gegenüber einem einzelnen Master-Knoten führen. Darüber hinaus kommt es zu einer Verbesserung der Verfügbarkeit, da der Master im Falle eines Ausfalls durch eine Replik ersetzt werden kann, die dadurch selbst zum Master wird. Den Ausfall des Masters bemerken die Repliken selbstständig.

Ein Nutzer von Hyrise-R könnte Anforderungen an den minimalen Replikationsgrad der Datenbank haben. Diese Anforderungen könnten durch eine Policy ausgedrückt werden. Gemäß den vorigen Policy-Beispielen läge es in der Verantwortung von OpenStack, die Policy umzusetzen.

[18] hat im Rahmen eines Masterprojektes oben beschriebene Infrastruktur auf OpenStack umgesetzt. Dazu wurden verschiedene Abbilder erstellt, die als VMs in OpenStack ausführbar sind. Nach dem Start der Dispatcher-VM können mehrere Hyrise-VMs gestartet werden. Eine zentrale Herausforderung besteht im Aufbau der Verbindungen zwischen den Hyrise-VMs untereinander sowie zum Dispatcher – dazu werden nämlich die Internet Protocol (IP)-Adressen der gewünschten Kommunikationspartner benötigt. Eine Lösung dazu wurde in einen *Cluster-Manager* integriert, der als eine weitere VM bereitgestellt wird. Der Cluster-Manager bietet eine Weboberfläche, über die unter anderem der Dispatcher und Hyrise-Instanzen erzeugt werden können. Nachdem OpenStack die über den Cluster-Manager angeforderten VMs gestartet hat und in Kenntnis ihrer IP-Adressen ist, loggt sich dieser dort per Secure Shell (SSH) ein und startet die eigentlichen Applikationen mit den nötigen Konfigurationseinstellungen inklusive der IPs.

Es wird ersichtlich, dass OpenStack die Umsetzung einer Policy mit Replikationsanforderungen im gewählten Hyrise-R-Kontext mittels virtuellen Maschinen vornehmen muss. Das entspricht der Kernkompetenz von OpenStack, erscheint somit gut umsetzbar. Während der Cluster-Manager zur manuellen Erstellung von Hyrise-R-Instanzen gut geeignet sein mag, ist er in einer mittels Policies automatisierten Umgebung fehlplatziert: Wenn eine Policy zur Replikation von Hyrise-R-Instanzen angeboten wird, erwarten Nutzer auch eine vollständig eingerichtete und untereinander verbundene Hyrise-Plattform; nicht aber weitgehend leere VMs, auf denen die Datenbank erst über eine zusätzliche Weboberfläche gestartet werden muss.

Daraus folgen zwei Möglichkeiten: Entweder OpenStack übernimmt die Aufgaben des Cluster-Managers und sorgt somit für die Einrichtung der VMs und ihrer Verbindungen untereinander oder aber die Policy muss in ihrer angestrebten Funktionalität eingeschränkt werden.

Wir betrachten zunächst die erste Möglichkeit. [18] nutzt zur Konfiguration der VMs SSH. Dies gehört nicht in den Aufgabenbereich von OpenStack. Es müssten daher andere Wege gefunden werden, die Hyrise-Applikation beziehungsweise den Dispatcher zu starten. Hilfreich wäre beispielsweise ein Skript, das sich in die VMs einspeisen lässt, dort beim Start ausgeführt wird und dabei Hyrise startet. Das lässt sich tatsächlich mit Bordmitteln von

OpenStack umsetzen. Die Nova-API[29] bietet zwei vielversprechende Wege, um eine VM mit einem dynamisch generierten Skript auszustatten:

- Die Anfrage zur Erstellung eines Servers darf ein optionales Feld `user_data` enthalten. Sein Wert muss eine Base64-kodierte Zeichenkette sein, die den Inhalt einer Datei repräsentiert, die beim Start der VM ausgeführt wird. Es folgt ein kurzes Beispiel. Eine VM soll folgendes Skript ausführen, was eine Datei mit dem Inhalt `foo` unter `/bar.txt` anlegt:

  ```
  #!/bin/sh

  echo "foo" > /bar.txt
  ```

  Eine valide zur Ausführung dieser Datei geeignete `POST`-Anfrage an die Nova-API unter dem Pfad `/servers` könnte folgenden Inhalt haben:

  ```
  1  {
  2      "server": {
  3          "name": "VM_with_injected_script",
  4          "imageRef": "ea1df6c9-9d68-42b7-906d-58b47f2b1ca9",
  5          "flavorRef": "1",
  6          "user_data": "IyEvYmluL3NoCgplY2hvICJmb28iID4gL2Jhci50eHQ=",
  7          "max_count": 1,
  8          "min_count": 1
  9      }
  10 }
  ```

  Die Zeilen 4 und 5 enthalten IDs zur Referenzierung eines Abbilds, von dem eine VM gestartet werden soll, sowie eines Flavors (siehe Abschnitt 7.4.1). Zeile 6 enthält obiges Skript in Base64-kodierter Form. Die Zeilen 7 und 8 sorgen dafür, dass genau eine VM angelegt wird. Diese Parameter werden später noch genauer betrachtet.

  Dieser Weg zur Injektion eines Skripts mittel `user_data` wurde erfolgreich ausprobiert. Er ist somit technisch geeignet, den Cluster-Manager zu ersetzen und das Starten der Hyrise-Applikationen OpenStack zu überlassen. [18] war diese Methode bekannt, hatte damit aber aus unbekannten Gründen keinen Erfolg.

- Als mögliche Alternative zu obigem `user_data`-Feld gibt es ein weiteres optionales Feld namens `personality`. Darüber kann eine Liste an Dateisystempfaden zur Bestimmung von Dateien sowie die zugehörigen Inhalte angegeben werden. Die Dateiinhalte müssen auch hier Base64-kodiert vorliegen. Beim Start der VM werden die Dateien eingeschleust. Dieser Weg wäre hilfreich, wenn Hyrise die benötigten Konfigurationsdaten wie zum Beispiel die IP des Dispatchers auch aus Dateien auslesen könnte. Da der erste Weg bereits funktionierte, wurde die Methode mittels des `personality`-Feldes nicht evaluiert.

Auch wenn eine technische Lösung zur Umsetzung des Hyrise-Starts durch OpenStack gefunden wurde, fiel die Entscheidung gegen eine Implementierung ebendieser. Der Grund

[29] `http://developer.openstack.org/api-ref/compute/?expanded=create-server-detail#create-server`

dafür ist, dass OpenStack für derlei Aufgaben nicht zuständig ist. OpenStack bietet IaaS; im beschriebenen Szenario handelt es sich jedoch um einen Anwendungsfall von PaaS: Die Ausführung von Hyrise geschieht nicht zum Selbstzweck, sondern die Datenbank dient einer anderen Applikation – Hyrise stellt eine Plattform für diese Applikation dar.

Würde OpenStack die Konfiguration von Hyrise vornehmen, würden die Trennung zwischen IaaS und PaaS durchbrochen. Die Felder `user_data` und `personality` sind daher für Nutzer vorgesehen, zur Verwendung durch OpenStack selbst jedoch eher nicht. Die reine Initiierung der Dateieinschleusung durch OpenStack beziehungsweise eine PolicyExtension wäre noch vertretbar, schließlich dient es der Bereitstellung einer Infrastruktur für die darauf ausgeführte Plattform. Da der Inhalt der Dateien jedoch dynamisch erstellt werden muss, um die durch OpenStack vergebenen IP-Adressen der anderen Hyrise-Instanzen zu enthalten, würde OpenStack damit in die Konfiguration der Plattform eingreifen. Dies entspricht nicht mehr den Zuständigkeiten OpenStacks.

Als Konsequenz ist die auf Hyrise bezogene Policy in der bisher betrachteten Art nicht umsetzbar. Daher erfolgte eine Anpassung der in diesem Beispiel umzusetzenden Policy: Replikation ist nach wie vor der Fokus, wird aber auf VMs beschränkt. Die Konfiguration der darin laufenden Anwendungen hat eine andere Partei zu verantworten. Denkbar wäre beispielsweise eine Komponente auf Plattformebene, die ebenfalls Policies interpretiert, aber nur für jene Teile zuständig ist, die sich auf Applikationen beziehen. Im Falle von Hyrise ginge es auch ohne eine solche Komponente, wenn Hyrise Mechanismen zur sogenannten *Service Discovery* nutzen würde. Dafür gibt es eine Vielzahl von Protokollen [4]. Auch für die im Cloud-Kontext populäre Microservice-Architektur existieren mehrere Entwurfsmuster zur Adressierung der Service Discovery.[30] Weder die Evaluierung der verfügbaren Lösungen noch die zur Umsetzung nötigen Anpassungen an Hyrise sind Thema dieser Arbeit.

Mit der erfolgten Einschränkung der Policy auf die Replikation von VMs sollen Nutzer in der Lage sein auszudrücken, von welchem Abbild wie viele Instanzen gestartet werden sollen. Das wirft eine Reihe von Fragen auf:

- Muss die in der Policy angegebene Anzahl an VMs vom Nutzer explizit angefordert werden oder erstellt OpenStack selbstständig diese Menge an Instanzen?
- Was passiert, wenn der Nutzer weniger oder mehr Instanzen anfordert? Ist die Anzahl in der Policy also ein Minimum, Maximum oder ein exakter Wert, von dem keine Abweichung erlaubt ist?
- Was passiert, wenn nachträglich eine Instanz desselben Abbilds durch denselben Nutzer gestartet oder gelöscht wird? Ist das erlaubt? Muss die Anzahl aus der Policy durch OpenStack immer gewährleistet sein, sodass auch Aktionen des Nutzers ausgeglichen werden? Bezieht sich die Zahl in der Policy also nur auf den Start einer VM oder auf deren gesamten Lebensdauer?
- Wie wird mit VMs verfahren, die nicht aus Abbildern erstellt werden, sondern aus in Cinder abgelegten virtuellen Festplatten? Generell: Ist eine solche Policy überhaupt kompatibel zur kompletten Funktionalität in Nova oder treten Inkompatibilitäten auf?

[30] `http://microservices.io/patterns/index.html`

> Oder anders gefragt: Soll die Umsetzung der Policy mit allen Randbedingungen, die beim Starten von VMs auftreten können, umgehen können?

Zur Beantwortung dieser Fragen wurden zwei Entscheidungen gefällt: Die bisherigen Policy-Beispiele unterbinden gegen die Policy verstoßende Anfragen. Um die Möglichkeiten des policyextension-Frameworks zu zeigen, soll dieses Beispiel schreibend auf eingehende Anfragen zugreifen, diese also falls nötig manipulieren. Des Weiteren liegt keine Priorität darauf, eine vollständige Integration des durch eine solche Policy entstehenden Features in OpenStack zu gewährleisten. Vielmehr soll ein übersichtliches Beispiel für eine schreibende Policy entstehen, das in Kauf nimmt, nur unter bestimmten Bedingungen zu funktionieren. Unter diesen Gesichtspunkten wurden viele der obigen Fragen willkürlich mit Fokus auf Einfachheit beantwortet. Das heißt zugleich aber nicht, dass sämtliche Konzepte zur sauberen Konstruktion von Software missachtet werden sollen, sondern vielmehr, dass, wenn nötig, Kompromisse zugunsten der Einfachheit eingegangen werden können.

Konkret bedeutet das, dass die Policy sich nur auf den Start von VMs bezieht. Mindestens die in der Policy angegebene Anzahl an Instanzen wird gestartet, unabhängig davon, ob der Nutzer bereits andere Instanzen desselben Abbilds ausführt. Falls der Nutzer explizit mehr Instanzen anfordert als in der Policy angegeben, so soll die explizite Anzahl ausschlaggebend sein. Weiterhin soll nur die Erstellung von VMs aus Abbildern betrachtet werden; die Möglichkeit zur Instanziierung eines Cinder-Datenträgers bleibt unberücksichtigt.

### 7.5.2 Umsetzung der Policy

Auch bei diesem dritten und letzten Policy-Beispiel soll die Umsetzung unter anderem anhand von Quellcode erläutert werden. Dabei werden konzeptionelle Grenzen des policy-frameworks verdeutlicht.

Da die Policy nicht etwa für alle zur Auswahl stehenden Abbilder eine Mindestreplikationsrate vorschreibt, sondern dies für einzelne Abbilder festlegbar sein soll, muss die Policy mit dem jeweiligen Abbild verknüpft werden. Um eine eindeutige Identifizierung zu gewährleisten, bietet sich die durch OpenStack vergebene und einsehbare ID eines Abbilds zur Referenzierung an. Das hat jedoch den Nachteil, dass bei Änderung eines Abbilds, was durch Löschen und erneutes Anlegen geschieht, ebenfalls Anpassungen an der Policy nötig sind, um die ID des referenzierten Abbilds aktuell zu halten.

Als Alternative wurde in Betracht gezogen, die Teile einer Policy, die sich auf Abbilder beziehen, im für sie verantwortlichen Dienst namens *Glance* zu speichern. Glance bietet die Möglichkeit, Metadaten zu Abbildern zu hinterlegen. Die Metadaten könnten als Ablage für Teile der Policy dienen. Das hätte jedoch den Nachteil, dass die dort angegebene Mindestreplikationsrate für alle Nutzer gilt. Zwar könnte man für verschiedene Nutzer andere Werte angeben, aber das würde das Problem lediglich verlagern: Es müssten dann Referenzen zu Nutzern hergestellt werden, dabei war die Motivation zur Speicherung in Keystone die Vermeidung wartungsanfälliger Referenzen. Darüber hinaus müssten Nutzer die Berechtigung erhalten, Abbilder zu bearbeiten, um schreibend auf die Metadaten zugreifen zu können. Der größte Nachteil allerdings wäre die örtliche Aufteilung von Policies: Statt einer zentra-

len Speicherung ausschließlich in Keystone würde die Policy zusätzlich zu Teilen in Glance gespeichert werden. Dieser Ansatz ist wartungsintensiv und wurde daher nicht verfolgt – es bleibt bei der zentralen Speicherung in Keystone.

Um die Zusammenhänge für dieses Policy-Beispiel leichter nachvollziehen zu können, erfolgt die Referenzierung der Abbilder nicht über deren ID, sondern ihren Namen. Die dabei womöglich auftretende Uneindeutigkeit durch Namenskollisionen wird in Kauf genommen.

Unter der Annahme, dass in Glance ein Abbild namens `Hyrise` registriert ist, könnte eine für dieses Policy-Beispiel valide Policy aus Sicht einer PolicyExtension wie folgt aussehen:

```
{
    "images": {
        "replication": [
            {
                "rate": 3,
                "name": "Hyrise"
            }
        ]
    }
}
```

Ersichtlich wird, dass sich auf Glance beziehende Vorgaben unter dem Schlüssel `images` zu finden sind. Dessen Wert ist vom Python-Datentyp `dict`, der wiederum unter dem Schlüssel `replication` eine Liste enthält. Deren Elemente sind abermals vom Typ `dict` und enthalten zwei Schlüssel: Der Wert für `rate` gibt die Mindestreplikationsrate an, der Wert für `name` ist die Referenz auf das in Glance registrierte Abbild, auf das die Policy sich bezieht.

Es folgt der im Vergleich zu vorigen Policy-Beispielen umfangreichere Quellcode der Implementierung. Er befindet sich in der (zu Novas Quellverzeichnis relativen) Datei `nova/policyextensions/replicate_image.py`.

```
1  from policyextension import PolicyExtensionBase, PolicyIncompatibleRequest
2
3  from nova import exception, utils
4
5
6  class InstanceReplicationExtension(PolicyExtensionBase):
7      func_paths = ['nova.compute.api.API._create_instance']
8
9      def _create_instance(self, func_args, policy):
10
11         def check_requested_networks_compability():
12             if max_count == 1:
13                 # Run checks only when just 1 instance was requested,
14                 # otherwise OpenStack took care of that already.
15                 requested_networks = func_args['requested_networks']
16                 if requested_networks and replication_rate > 1:
17                     # Run checks only if policy requires more VMs.
18                     try:
19                         api._check_multiple_instances_and_specified_ip(
20                             requested_networks)
21                         if utils.is_neutron():
22                             api._check_multiple_instances_neutron_ports(
```

```
23                              requested_networks)
24                  except (exception.InvalidFixedIpAndMaxCountRequest,
25                          exception.MultiplePortsNotApplicable):
26                      msg = ("Your policy requires redundancy for %s. "
27                             "This results in spawning multiple identical "
28                             "instances. This is not compatible with "
29                             "requesting a fixed IP or a specified port ID.")\
30                             % image_name
31                      raise PolicyIncompatibleRequest(msg)
32
33          image_href = func_args['image_href']
34          if not image_href:
35              # We only deal with starting instances from images. This instead
36              # seems to be a request to start an instance from a volume.
37              return
38          api = func_args['self']
39          image_name = api.image_api.get(func_args['context'], image_href)['name']
40          max_count = func_args['max_count'] or 1
41          max_count = int(max_count)
42          try:
43              for image in policy['images']['replication']:
44                  if image['name'] == image_name:
45                      replication_rate = int(image['rate'])
46
47                      # For a single instance it is possible to request a fixed IP
48                      # or a specific port ID. If we modify the creation request
49                      # to spawn more instances, they cannot share the same IP or
50                      # port ID, therefore those options are not compatible with
51                      # this PolicyExtension.
52                      check_requested_networks_compability()
53
54                      func_args['max_count'] = max(replication_rate, max_count)
55                      func_args['min_count'] = func_args['max_count']
56
57                      break
58          except (KeyError, TypeError, ValueError):
59              pass
```

Der Quellcode soll im Folgenden erläutert werden, allerdings nicht komplett in der Reihenfolge der Zeilen:

- Zeile 1 importiert die bekannte Oberklasse einer PolicyExtension und `PolicyIncompatibleRequest` – eine Fehlerklasse, auf die später genauer eingegangen wird.
- Zeile 3 importiert hilfreiche, zu Nova gehörende Module.
- Nachdem Zeile 6 die eigentliche PolicyExtension-Klasse definiert, wird in Zeile 7 die um Funktionalität zu ergänzende Originalfunktion angegeben. Es sei darauf hingewiesen, dass bereits das zweite Policy-Beispiel den Start von VMs betrachtete (siehe Abschnitt 7.4.3). Dort wurde in `func_paths` allerdings eine andere Originalfunktion benutzt. Der Grund dafür liegt in der Validierung der eingehenden Anfrage: In der obersten Schicht der API werden noch nicht alle Bestandteile der Anfrage validiert, statt-

dessen findet die Validierung auch zu Teilen in darunterliegenden Schichten statt. Für dieses Policy-Beispiel werden die benötigten Daten in einer tieferen Schicht validiert, sodass eine andere Originalfunktion gewählt wurde.

Bei der Auswahl einer geeigneten Originalfunktion, vor der zusätzlich eigener Code ausgeführt werden soll, ist daher genau zu prüfen, ob alle zur Umsetzung der Policy benötigten Daten dort zur Verfügung stehen, in diesem Fall beispielsweise die Anzahl der vom Nutzer angeforderten Instanzen sowie das ausgewählte Abbild. Darüber hinaus sollten diese Daten bereits durch OpenStack validiert worden sein. Andernfalls muss die Validierung in der PolicyExtension selbst vorgenommen werden, was jedoch Codeduplikation zur Folge hätte.

- Zeile 9 definiert eine Funktion, die den gleichen Namen wie die Originalfunktion trägt. Auf die in Zeile 11 folgende geschachtelte Funktion wird später eingegangen.
- Zeile 33 liest das vom Nutzer angegebene Abbild, von dem eine oder mehrere VMs erstellt werden sollen.
- Sollte der Nutzer kein Abbild angegeben haben, so geht die PolicyExtension davon aus, dass der Nutzer eine VM auf Basis einer in Cinder gespeicherten Festplatte erstellen will. Die Zeilen 34 bis 37 sorgen dafür, dass die PolicyExtension dann nichts unternimmt, sie also nur Einfluss auf die Erstellung von VMs aus Abbildern hat.
- Zeile 38 enthält eine Besonderheit: Die Originalfunktion ist eine Methode einer Klasse namens `API`. Da die folgende Zeile auf eine Referenz derselben Objektinstanz, zu der auch die Originalfunktion gehört, zugreifen muss, wird dieses spezifische `API`-Objekt extrahiert und der lokalen Variable `api` zugewiesen.
- Zeile 39 nutzt das `API`-Objekt und eine der Methoden des darin referenzierten Objekts, um aus der vom Nutzer angegebenen Referenz auf das Abbild dessen Namen zu ermitteln.
- Die Nova-API erlaubt, mehrere Instanzen mit einer einzigen Anfrage anzufordern. Dazu gibt es die optionalen Felder `min_count` (standardmäßig 1) sowie `max_count` (standardmäßig wertgleich mit `min_count`). Die Zeilen 40 und 41 extrahieren daher `max_count` aus den vom Nutzer angegebenen Daten oder setzen den Wert auf 1, falls der Nutzer das Feld leer lässt.
- Die Zeilen 43 und 44 lesen die Policy aus. Dazu iterieren sie über alle darin angegebenen Replikationsanforderungen und prüfen, ob Vorgaben für das vom Nutzer ausgewählte Abbild enthalten sind.
- Falls ja, dann wird die dort angegebene Mindestreplikationsrate in Zeile 45 ausgelesen.
- Der Aufruf der Funktion in Zeile 52 wird später erläutert.
- Die Zeilen 54 und 55 sind die entscheidende Stelle, an der die PolicyExtension womöglich manipulierend eingreift: Ist die vom Nutzer angeforderte Anzahl an Instanzen kleiner als durch die Policy vorgegeben, so werden die Werte für `max_count` und `min_count` ersetzt durch die in der Policy angegebene Mindestreplikationsrate.

- Zeile 57 bricht die Iteration über die Policy ab, falls für das zu startende Abbild Vorgaben gefunden wurden. Sollte die Policy zum gewählten Abbild keine Vorgaben enthalten, hat die PolicyExtension keinen Effekt.
- Sollte die Policy nicht einmal Angaben zu Abbildern generell oder zu Replikationsraten enthalten, so sorgen die Zeilen 58 und 59 für die normale weitere Bearbeitung der Anfrage durch OpenStack, ohne dass die PolicyExtension Auswirkungen hat.

Noch unerläutert geblieben ist die geschachtelte Funktion `check_requested_networks_compability`. Die Nova-API zur Erstellung von VMs bietet die Funktionalität, eine spezifische IP-Adresse beziehungsweise einen spezifischen virtuellen Netzwerkanschluss anzufordern. Dies ist allerdings nur möglich, wenn nur eine einzelne Instanz gefordert wird, da sich nicht mehrere VMs dieselbe IP-Adresse oder denselben Netzwerkanschluss teilen können. Diese Funktionalität ist daher mit einer Policy, die eine Mindestreplikationsrate $> 1$ vorschreibt, nicht kompatibel.

OpenStack enthält Logik, um mit dem Fall umzugehen, dass sowohl mehrere Instanzen als auch obige Netzwerkeigenschaften angefordert werden. Es stellt sich also die Frage, warum die PolicyExtension die Originalfunktion nicht so wählt, dass sie `max_count` und `min_count` überschreiben kann und dies anschließend von der vorhandenen Logik auf Konflikte geprüft wird.

Das wäre in der Tat möglich und würde obigen Quellcode deutlich kürzen. Es hätte allerdings den Nachteil, dass der Nutzer womöglich die Fehlermeldung erhält, `max_count` dürfe nicht größer als eins sein, wenn eine feste IP-Adresse angefordert wird. Nutzer ändern ihre Policy wahrscheinlich nur selten. Zusätzlich liegt sie wenig präsent in Keystone. Falls ein Nutzer sich der Existenz seiner Policy nicht bewusst ist und nur eine VM anfordert, könnte eine solche Fehlermeldung zu Verwirrung führen. Wünschenswert wäre hingegen, dass die Fehlermeldung einen Hinweis enthält, dass sie durch die Policy verursacht wurde, also auf die Inkompatibilität zwischen mehreren Instanzen und obigen Netzwerkeinstellungen aufmerksam macht. Aus diesem Grunde ruft die Funktion `check_requested_networks_compability` die vorhandene Logik zur Feststellung eines solchen Konflikts selbst auf. Im Falle eines Konflikts erhält der Nutzer statt einer unverständlichen Nachricht eine aussagekräftige Fehlermeldung. Dafür wird die durch das policyextension-Framework bereitgestellte Klasse `PolicyIncompatibleRequest` genutzt.

Letztlich kommt es damit zu Codeduplikation, weil die Konflikterkennungslogik sowohl durch die PolicyExtension als auch durch OpenStack selbst während der Verarbeitung der Anfrage aufgerufen wird. Dies liegt an der Einschränkung durch das policyextension-Framework, das im aktuellen Zustand den Code einer PolicyExtension lediglich zeitlich vor einer Originalfunktion ausführen kann. Die Diskussion über Stärken und Schwächen des Frameworks in Abschnitt 7.8 greift diesen Punkt auf.

### 7.5.3 Evaluierung

Nachdem die PolicyExtension bereits duplizierten Code benötigte, um dem Nutzer unerwartete Fehlermeldungen zu ersparen, müssen auch weitere durch die Manipulation von Daten entstehenden Folgen evaluiert werden.

Der schreibende Zugriff auf die Anfrage ist auf der einen Seite ein sehr mächtiges Mittel: Während die vorigen Policy-Beispiele lediglich Fehlermeldungen aufgrund von Verstößen zurückgaben und der Nutzer somit in der Pflicht war, eine neue, korrigierte Anfrage zu senden, so kann die Manipulation dem Nutzer diese Arbeit abzunehmen, indem die Anfrage automatisch korrigiert wird.

Auf der anderen Seite können die manipulierten Anfragen zu unerwarteten Ergebnissen führen. Dazu gehört beispielsweise das oben erläuterte Konstrukt für verständlichere Fehlermeldungen, aber die Manipulation könnte auch Auswirkungen auf die Antworten von erfolgreich verarbeiteten Anfragen haben. Die von OpenStack zurückgegebene Antwort nach dem erfolgreichen Anlegen einer einzelnen Instanz unterscheidet sich zwar nicht von jener mit mehreren Instanzen, grundsätzlich ist ein solcher Unterschied aber denkbar. Ein menschlicher Nutzer mag dann in der Lage sein, mit der teils unerwarteten Antwort umzugehen – automatisierte Werkzeuge sind es womöglich nicht. Novas Kommandozeilentool beispielsweise erhält als Antwort auf die Anfrage zur VM-Erstellung einen URL, über den der Fortschritt der Aktion verfolgt werden kann. Wenn mit dem Tool nur eine Instanz angefordert wurde, werden nach Erfolg auch nur die Details zu einer VM angezeigt. Ob weitere Instanzen erstellt wurden, erfährt der Nutzer überhaupt nicht.

Darüber hinaus kann es zu Kollisionen mit Novas Kontingentsystem kommen: Nutzern ist ein gewisses Kontingent an möglichen VMs zugewiesen. Während eine unveränderte Anfrage dieses womöglich nicht verletzt, kann die Manipulation hingegen zu einer Überschreitung führen. Dieser mögliche Fehler wurde in der PolicyExtension bislang nicht beachtet. Davon betroffen ist beispielsweise OpenStacks webbasiertes Dashboard: Es lässt das Anlegen neuer Instanzen überhaupt nur innerhalb des Kontingents zu. Von der serverseitigen Manipulation weiß es jedoch nichts. Das führt dazu, dass beim Auftritt eines kontingentbezogenen Fehlers die im Dashboard angezeigte Fehlermeldung für den Nutzer unverständlich ist: Obwohl das Dashboard von Nova die Nachricht *Quota exceeded for instances* erhält, wird dem Nutzer lediglich *Error: Unable to create the server* angezeigt. Details zur Fehlerbehandlung im Dashboard folgen in Abschnitt 8.2.

Aufgrund der vielen Nachteile und Nebenwirkungen, die durch die Manipulation möglich werden, ist im Allgemeinen eher von manipulierenden PolicyExtensions abzuraten – auch wenn es sicher valide Anwendungsfälle gibt. Stattdessen sind eher Umsetzungen von Policies zu empfehlen, die Verstöße auch kommunizieren statt sie still beheben zu wollen.

Von den drei Policy-Beispielen scheint am ehesten das erste zur Festplattenverschlüsselung für eine manipulierende PolicyExtension geeignet. Die Verschlüsselung passiert ohnehin transparent auf Serverseite. Von daher wäre denkbar, dass die PolicyExtension selbstständig einen für Verschlüsselung konfigurierten VolumeType wählt oder diesen sogar anlegt, falls noch keiner existiert.

### 7.5.4 Entwicklungsumgebung

Obwohl sich auch dieses Policy-Beispiel auf Nova bezieht, ist die aus Abschnitt 7.4.2 bekannte Entwicklungsumgebung nicht ausreichend. Dort genügte es, als Entwickler durch mutwilligen Verstoß gegen die in Keystone hinterlegte Policy eine durch die PolicyExtension ausgelöste PolicyViolation zu Gesicht zu bekommen. Dazu musste lediglich Novas API lauffähig sein, die anderen zu Nova gehörenden Prozesse hingegen nicht. Das hatte zwar zur Folge, dass keine VM tatsächlich gestartet werden konnte, aber zur Ausgabe der PolicyViolation genügte es.

Für dieses Policy-Beispiel hingegen wäre wünschenswert zu sehen, wie tatsächlich mehr VMs erstellt werden, als tatsächlich angefordert wurden. Dafür müsste nicht nur Nova umfassend eingerichtet werden, auch Glance als Quelle für die Abbilder wäre erforderlich. Einfacher ist die Nutzung von *DevStack*[31]. Dabei handelt es sich um ein Projekt, bei dem ein ganzes OpenStack auf einer einzigen Maschine beziehungsweise in einer einzelnen VM installiert wird. Es bezweckt nicht eine Einrichtung von OpenStack zur produktiven Nutzung, sondern richtet sich explizit an Entwickler, die eine möglichst kompakte OpenStack-Instanz benötigen.

Alle vorherigen Abschnitte zur Einrichtung von Entwicklungsumgebungen sind dadurch keinesfalls obsolet: Da DevStack bei der Installation unter anderem in die Netzwerkkonfiguration eingreift, bietet sich die Nutzung einer VM für DevStack an. Wenn man als Entwickler nicht innerhalb dieser VM entwickeln will, was sich in der Praxis als umständlich erwiesen hat, dann stellt sich die Frage, wie der lokale Quellcode in die DevStack-VM gelangt. Dazu muss eine Form der Synchronisation gewählt werden. Versionskontrollsysteme sind weniger geeignet, da die Funktionalität des Codes vermutlich erst überprüft werden soll, bevor eine neue Revision erstellt wird. Die Vielzahl der zu synchronisierenden Dateien der einzelnen OpenStack-Projekte führte zu einer unbefriedigenden Leistung der Synchronisation. Getestet wurden Network File System (NFS), SSH Filesystem (SSHFS) und die in die Virtualisierungslösung VMware Fusion integrierte Dateisystemsynchronisierung zwischen Virtualisierungshost und VM. Aus diesem Grunde ist es für Entwickler komfortabler, die Entwicklungsumgebungen auf der zur Entwicklung genutzten Maschine einzurichten, solange sich der Aufwand dafür in Grenzen hält. Das war bei den bisherigen Entwicklungsumgebungen gegeben, in diesem Fall jedoch ist der Einsatz von DevStack sinnvoller. Die Einrichtung einer DevStack-Instanz samt Integration der drei Policy-Beispiele wird in Kapitel 9 genauer betrachtet.

## 7.6 Unabhängigkeit vom Format der Policy durch PolicyFormat

Da CPPL im Rahmen des SSICLOPS-Projekts als Policy-Sprache mit Fokus auf Einsatz in Cloud-Umgebungen entworfen wurde, wäre die Nutzung von CPPL in dieser Arbeit wünschenswert gewesen. Weil die Fortschritte an CPPL bislang allerdings vor allem konzeptioneller Natur sind, wurde in dieser Arbeit und in den vorigen Beispielen mit Pythons Datentyp `dict` gearbeitet.

[31] `http://docs.openstack.org/developer/devstack/`

Um eine nachträgliche Integration von CPPL möglichst leicht zu gestalten, ist das policyframework nahezu unabhängig vom konkreten Format einer Policy. Die einzige Voraussetzung ist bislang, dass die Policy zum Zeitpunkt der Auswertung in einer PolicyExtension als `dict` verfügbar sein muss. Diese Anforderung war bereits implizit bei den Policy-Beispielen ersichtlich, als jeweils gezeigt wurde, wie eine valide Policy aus Sicht der PolicyExtensions auszusehen hat.

Dazu bietet das policyextension-Framework die abstrakte Klasse `PolicyFormat` an. Eine nicht-abstrakte Unterklasse ist verantwortlich für die Konvertierung einer Policy zwischen `dict`, dem Format der Speicherung in Keystone sowie einer textuellen Ausgabe für den Nutzer. Eine solche Unterklasse muss mehrere Methoden implementieren, die im Folgenden mit Bezug zu CPPL erläutert werden:

- `parse(self, textual_policy)`: Die Methode erwartet die Policy in einer textuellen Form, wie sie beispielsweise durch einen Nutzer formuliert und eingegeben wird. Dies entspricht der nicht-binären Form von CPPL. Die Policy wird als `dict` zurückgegeben. Falls die Policy nicht geparst werden kann, muss ein Fehler der Klasse `PolicyIncompatibleFormat` geworfen werden.
- `to_text(self, policy)`: Die Methode stellt die Rückrichtung dar – eine Policy des `dict`-Typs wird textuell zurückgegeben, sodass sie für Nutzer verständlich ist. `PolicyIncompatibleFormat` muss geworfen werden, wenn die Konvertierung nicht möglich ist.
- `get_empty_textual_policy(self)`: Die Methode muss die textuelle Darstellung einer leeren Policy zurückgeben. Somit gilt: $parse(get\_empty\_textual\_policy()) == dict()$.
- `encode(self, policy)`: Die Methode erwartet eine Policy als `dict` und gibt sie in einer Form zurück, die zur Speicherung in Keystone vorgesehen ist. Die Rückgabe entspricht der binären Form von CPPL. Fehler führen zum Werfen von `PolicyIncompatibleFormat`.
- `decode(self, policy)`: Diese Methode ist die Rückrichtung zu `encode` – sie nimmt eine Policy entgehen, wie sie in Keystone gespeichert wird und gibt sie als `dict` zurück. Bei Fehlern wird `PolicyIncompatibleFormat` geworfen.
- `pretty_print(self, policy)`: Die Implementierung dieser Methode ist optional, sie ist standardmäßig identisch zu `to_text`. Wird sie implementiert, so soll sie eine visuell ansprechende textuelle Form der Policy ausgeben, sofern `to_text` dies nicht leistet.

Es gilt somit für eine textuelle Policy *textual_policy*:
$to\_text(decode(encode(parse(textual\_policy)))) == textual\_policy$
Das policyextension-Framework enthält bereits zwei beispielhafte Implementierungen für Unterklassen von `PolicyFormat`:

1. `JSONFormat`: Dieses Format bietet sich durch die Ähnlichkeit von JSON zu `dict` an. Policies werden daher in Keystone als JSON gespeichert und auch die textuelle Darstellung für Nutzer erfolgt als JSON.

Deutlich werden hier die Unterschiede zwischen `to_text` und `pretty_print`: Während `pretty_print` das JSON mit Einrückungen und Zeilenumbrüchen darstellt, fehlt der Ausgabe von `to_text` diese visuelle Unterstützung.

Auch die Anwendung von `get_empty_textual_policy` wird deutlich: Wenn ein Nutzer zum ersten Mal seine Policy bearbeitet und bisher keine Policy hatte, sollte etwas angezeigt werden, was im Idealfall bereits einen Hinweis auf das benötigte Format enthält – eine leere JSON-Zeichenkette wäre für dieses Beispiel daher angemessen. `get_empty_textual_policy` gibt demnach `'{}'` zurück.

2. `JSONZipFormat`: Dieses Format demonstriert die Fähigkeit, mit binären Policies umzugehen, wie beispielsweise CPPL. Dazu erbt es von `JSONFormat`, überschreibt aber `encode` und `decode`, sodass das textuelle JSON vor Speicherung in Keystone gezippt und entsprechend auch wieder entpackt wird.

Unklar ist noch, wie angegeben wird, welche Unterklasse von `PolicyFormat` genutzt werden soll. Standardmäßig greift das policyextension-Framework auf `JSONFormat` zurück. Das ist zwar weniger kompakt als `JSONZipFormat`, erleichtert aber während der Entwicklung die Fehlersuche durch die Speicherung in einer für Menschen lesbaren Form.

In den Policy-Beispielen wurde gezeigt, dass die um Unterstützung für Policies erweiterten OpenStack-Dienste nur an einer einzigen Stelle angepasst werden mussten: Das Laden der PolicyExtensions aus einem Verzeichnis musste hinzugefügt werden. Dies geschah mit der Funktion `load_from` des Frameworks. Dieser lässt sich neben dem Verzeichnis der PolicyExtensions auch ein zu nutzendes `PolicyFormat` über das Argument `policy_format` mitgeben. Die für die Policy-Beispiele angepasste Datei `service.py` hätte somit alternativ auch um folgende Zeilen ergänzt werden können:

```
import policyextension
from policyextension.formats import JSONZipFormat

policy_dir = os.path.join(os.path.dirname(__file__), 'policyextensions')
policyextension.load_from(policy_dir, policy_format=JSONZipFormat)
```

Ein beliebiges Austauschen des genutzten Formats ist nicht ohne Weiteres möglich, sobald Policies in Keystone abgelegt wurden. Diese könnten dann nicht mehr dekodiert werden, sofern das genutzte Format nicht speziell auf die Präsenz des veralteten Formats vorbereitet ist. Grundsätzlich sind Migrationen zwischen Formaten mittels darauf vorbereiteter Implementierungen von `PolicyFormat` aber durchaus möglich.

## 7.7 Features und Interna des policyextension-Frameworks

Der Aufbau einer PolicyExtension mithilfe des policyextension-Frameworks ist bereits durch Beispiele bekannt, ebenso die Unabhängigkeit vom Format einer Policy durch PolicyFormat. Das Framework bietet darüber hinaus einige weitere Features, die zu Beginn dieses Abschnitts zusammen mit Details bereits bekannter Features vorgestellt werden. Für Beispiele zu den neuen Features sei auf den Quellcode des policyextension-Frameworks verwiesen: Neben

der eigentlichen Logik des Frameworks sind dort beispielhafte PolicyExtensions enthalten, die alle Features abdecken. Im Anschluss wird die generelle Funktionsweise erklärt, mit der Code aus den PolicyExtensions in OpenStack zur Ausführung gebracht wird. Abschließend werden die dabei aufgetretenen Herausforderungen samt ihrer Lösungen betrachtet.

### 7.7.1 Ausführungsreihenfolge mittels Prioritäten

Das policyextension-Framework muss mit dem Fall umgehen können, dass sich zwei oder mehr PolicyExtensions auf dieselbe Originalfunktion beziehen, also ein gemeinsames Element in ihrer `func_paths`-Liste haben. Dabei sollen alle PolicyExtensions ausgeführt werden können, trotz der gemeinsamen Originalfunktion. Diese Anforderung wird durch die Art und Weise, wie die Erweiterungen ausgeführt werden, standardmäßig erfüllt.

Schwieriger ist es, falls die PolicyExtensions in einer gewissen Reihenfolge ausgeführt werden müssen. Denkbar wäre beispielsweise, dass eine Erweiterung schreibend auf den Header einer Anfrage zugreift und dort ein Feld hinzufügt, während eine andere Erweiterung dieses neue Feld voraussetzt. Die Python-Funktionalität zum Laden der PolicyExtensions kann keine Reihenfolge garantieren. Aus diesem Grund kann eine PolicyExtension eine Priorität über das Attribut `priority` angeben, die vom Framework berücksichtigt wird. `priority` ist optional, muss bei Nutzung aber einen ganzzahligen Wert haben. Je höher die angegebene Priorität, desto früher läuft der Code der Erweiterung. In obigem Beispiel müsste daher die Erweiterung, die dem Header ein Feld hinzufügt, eine höhere Priorität haben als jene, die auf das Feld lesend zugreift.

### 7.7.2 Fehlerklassen

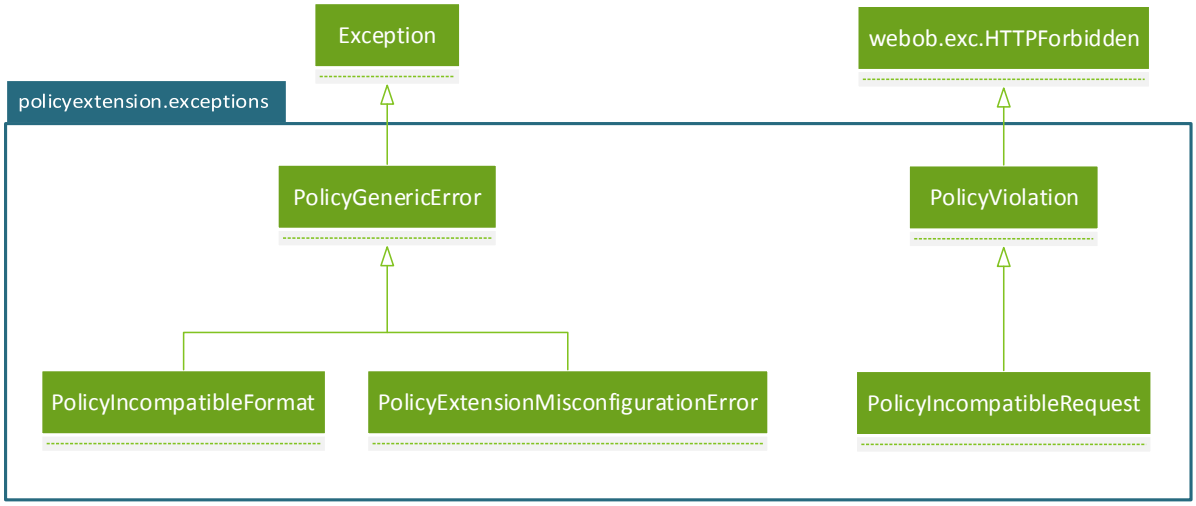


**Abbildung 7.3:** Klassendiagramm der durch das policyextension-Framework bereitgestellten Fehlerklassen. Das Framework nutzt sie teils intern selbst. Für PolicyExtensions sind vor allem `PolicyViolation` und `PolicyIncompatibleRequest` relevant.

Die Fehlerklasse `PolicyViolation` ist bereits aus den Policy-Beispielen bekannt. Es wird erwartet, dass beim Werfen eines solchen Fehlers die Anfrage abgebrochen und mit der angegebenen Fehlermeldung beantwortet wird. `PolicyViolation` stammt aus dem policyextension-Framework, ist daher OpenStack nicht bekannt. Der Auftritt eines solchen unbekannten Fehlers würde eigentlich durch eine Antwort mit dem HTTP-Statuscode 500 (Internal Server Error) quittiert werden. Damit könnte ein Nutzer wenig anfangen. Daher erbt `PolicyViolation` von der in OpenStack bekannten Fehlerklasse `HTTPForbidden`, die für eine Antwort mit dem HTTP-Statuscode 403 (Forbidden) steht. Die gleiche Problematik betrifft die aus Policy-Beispiel 3 (Abschnitt 7.5) bekannte Fehlerklasse `PolicyIncompatibleRequest`, die deshalb wiederum von `PolicyViolation` erbt.

Die vom PolicyFormat (Abschnitt 7.6) bekannte Fehlerklasse `PolicyIncompatibleFormat` hingegen erbt von `PolicyGenericError`, welches wiederum von Pythons allgemeiner `Exception`-Klasse erbt. Ein Auftreten von `PolicyIncompatibleFormat` bei einer Nutzeranfrage muss daher abgefangen und behandelt werden, wenn das Ergebnis kein HTTP-500-Fehler sein soll.

Des Weiteren existiert `PolicyExtensionMisconfigurationError`, welches ebenfalls von `PolicyGenericError` erbt. Die Fehlerklasse wird geworfen, wenn eine fehlerhafte Konfiguration einer PolicyExtension festgestellt wurde, beispielsweise wenn der Wert des `priority`-Attributes keine ganze Zahl ist. Abbildung 7.3 zeigt eine Übersicht der beschriebenen Fehlerklassen.

### 7.7.3 Versionsrestriktion

PolicyExtensions treffen Annahmen über OpenStack-Code, zum Beispiel mittels ihres `func_paths`-Attributs über das Vorhandensein von Funktionen. Gleichzeitig sind PolicyExtensions jedoch nicht Teil der offiziellen Codebasis von OpenStack. Eine neue OpenStack-Version kann daher inkompatibel zu den bestehenden PolicyExtensions sein, wenn beispielsweise Funktionen umbenannt oder entfernt wurden. Das policyextension-Framework bietet daher die Möglichkeit, das Laden einer PolicyExtension auf bestimmte Versionen eines Python-Paketes einzuschränken. Eine PolicyExtension für Cinder könnte beispielsweise eine Minimal- und Maximalversion für die installierte Version des Pakets `cinder` angeben. Minimal- und Maximalversion bilden ein abgeschlossenes Intervall. Entspricht die tatsächlich vorhandene Version nicht den Kriterien, so wird die Erweiterung nicht geladen.

Eine PolicyExtension kann die optionalen Attribute `min_ver`, `max_ver` und `package` nutzen. Ihre Werte müssen Zeichenketten sein. `package` muss gesetzt werden, falls `min_ver` oder `max_ver` vorhanden sind. Es genügt, entweder `min_ver` oder `max_ver` anzugeben. OpenStack nutzt semantische Versionierung[32], die Werte von `min_ver` und `max_ver` werden daher im `MAJOR.MINOR.PATCH`-Format erwartet, wobei nur die `MAJOR`-Angabe vorhanden sein muss.

[32] `http://semver.org/`

### 7.7.4 Ausführung der PolicyExtensions mittels Monkey Patching

Es ist bereits bekannt, dass der in einer PolicyExtension enthaltene Code vor jenem der angegebenen Originalfunktion ausgeführt wird. Im Folgenden wird erläutert, wie das policyextension-Framework dies bewerkstelligt.

In den drei Policy-Beispielen wurde gezeigt, das der originale OpenStack-Code nur um den Aufruf der Funktion `load_from` mit Angabe des PolicyExtension-Verzeichnisses erweitert werden muss. Die Funktion importiert alle im Verzeichnis enthaltenen Python-Dateien. Anschließend betrachtet es alle bekannten Unterklassen von `PolicyExtensionBase` – dort sind die gerade importierten PolicyExtensions enthalten. Entsprechend ihrer Prioritäten werden sie in der richtigen Reihenfolge instanziiert, wobei bereits eine Reihe von Überprüfungen vorgenommen wird, ob die Vorgaben zum Aufbau einer PolicyExtension eingehalten wurden. Danach erfolgt die Übergabe der Instanzen an ein `loader`-Modul, in dem zunächst weitere Überprüfungen stattfinden. Anhand der `func_paths`-Attribute werden die angegebenen OpenStack-Module geladen und überprüft, ob ihre Version ein weiteres Laden der jeweiligen PolicyExtension ausschließt. Es folgt der Hauptschritt: Die im `func_paths`-Attribut referenzierte Originalfunktion wird ausgetauscht. An ihre Stelle tritt eine Funktion, die zuerst den Code der PolicyExtension aufruft und dann die ehemalige Originalfunktion. Die Ersatzfunktion ist demnach ein zur Laufzeit konstruierter Wrapper der Originalfunktion, der den Aufruf der PolicyExtension beinhaltet. Das policyextension-Framework nutzt demnach sogenanntes *Monkey Patching*, nämlich Anpassungen an Modulen oder Klassen zur Laufzeit.

### 7.7.5 Bereitstellung der Policy durch Inspektion des Aufrufstapels

Eine Schwierigkeit stellt die Bereitstellung der Policy dar. Die Methoden einer PolicyExtension erhalten sie komfortabel über den Parameter `policy`. Um dies möglich zu machen, greift das Framework im Hintergrund zu ungewöhnlichen Maßnahmen.

Die Policy erreicht einen OpenStack-Dienst bekanntlich als Teil einer Anfrage, nämlich im HTTP-Header. In der obersten Softwareschicht der zum Dienst gehörenden API steht die eingegangene Anfrage als Objekt zur Verfügung, sodass dort die Policy ausgelesen werden kann. Während die Anfrage zwar zu weiteren Funktionsaufrufen und somit einer Bearbeitung in tieferen Schichten führt, so wird das eigentliche Anfrageobjekt aber nicht an diese tieferen Ebenen weitergereicht. PolicyExtensions können sich allerdings auf Originalfunktionen der unteren Softwareschichten beziehen. Dort ist die Policy durch das nicht durchgereichte Anfrageobjekt nicht vorhanden.

Um dieses Problem zu lösen, könnte der OpenStack-Code angepasst werden, sodass das Anfrageobjekt oder zumindest die daraus extrahierte Policy auch an die tieferen Schichten weitergegeben wird. Dieser Weg hätte diverse Nachteile, wie beispielsweise die weniger genaue Abgrenzung der einzelnen Ebenen voneinander oder die Unklarheit, wie tief ein solches Objekt durchgereicht wird. Letztlich läuft dieser Weg ohnehin den bisherigen Zielen zuwider: OpenStack soll nur minimal angepasst werden müssen. Das ist bisher gelungen, sodass hier ein anderer Weg gefunden werden muss.

Denkbar wäre, dass eine PolicyExtension neben der Originalfunktion auch jene höher liegende Funktion angeben muss, in der das Anfrageobjekt samt Policy noch verfügbar ist. Das Framework könnte in dieser obersten Schicht die Policy abgreifen und sie dann PolicyExtensions in tieferen Schichten zur Verfügung stellen.

Die tatsächlich umgesetzte Lösung geht noch einen Schritt weiter. PolicyExtensions sollen möglichst einfach bleiben, sodass diese zusätzlich Angabe vermieden werden sollte. In die tieferen Schichten der API-Implementierung eines Dienstes gelangt man über (geschachtelte) Funktionsaufrufe. Diese erfolgen synchron. Während eine Funktion in unteren Schichten ausgeführt wird, ist daher auch die Funktion in der obersten Schicht mit dem Anfrageobjekt samt Policy noch nicht beendet, sondern wartet auf Beendigung der Anfragebearbeitung auf den unteren Ebenen. Das heißt, dass die Policy während der Ausführung der unteren Schichten im Speicher verfügbar ist, lediglich nicht von der tiefen Ebene aus. Dies macht sich das Framework zunutze: Es läuft den Aufrufstapel hinauf bis zur obersten API-Ebene, extrahiert dort die Policy und stellt diese anschließend der PolicyExtension zur Verfügung. Das zum Zugriff auf den Aufrufstapel genutzte Modul `inspect` ist Teil von Pythons Standardbibliothek.

Das Erreichen der obersten Schicht erkennt das Framework am Vorhandensein einer bestimmten lokalen Variable, die das Anfrageobjekt enthält. Standardmäßig sucht es nach einer Variable namens `req`, der Name kann jedoch durch die PolicyExtension konfiguriert werden, indem diese dem Attribut `request_name` einen anderen Wert zuweist. Sollte kein Anfrageobjekt im gesamten Aufrufstapel gefunden werden, so wird ein Fehler der Klasse `PolicyGenericError` geworfen. Falls zwar die Anfrage gefunden wird, aber dort keine Policy angehängt ist, so wurde die policymiddleware nicht oder fehlerhaft konfiguriert. Auch darauf weist ein `PolicyGenericError` mit entsprechender Fehlermeldung hin.

### 7.7.6 Bereitstellung der Argumente der Originalfunktion

Neben dem `policy`-Parameter erhalten die Methoden einer PolicyExtension auch den Parameter `func_args`. Dieser ist bekanntlich vom Typ `dict` und enthält die Namen der Parameter der Originalfunktion sowie deren beim Aufruf übergebenen Werte. Diese Zugehörigkeit der Werte zu ihren Parameternamen macht die Verwendung in der PolicyExtension komfortabel, ist allerdings für das Framework nicht trivial.

Die vom policyextension-Framework installierte Ersatzfunktion erhält bei ihrem Aufruf eine Reihe von Argumenten, die eigentlich für die Originalfunktion gedacht sind. Einfach wäre es für das Framework, wenn es von den Erweiterungen verlangen würde, mit den Argumenten der Originalfunktion umgehen zu können. Die Folge dessen wäre eine von Erweiterung zu Erweiterung unterschiedliche Methodensignatur. Um die Erstellung von PolicyExtensions aber einfach zu halten, wurde entschieden, dass die Methoden einer Erweiterung eine gemeinsame Methodensignatur haben sollen; dazu werden die Werte der Argumente mittels `func_args` zur Verfügung gestellt.

Problematisch ist, dass die für die Originalfunktion gedachten Parameter nicht zwingend samt zugehörigem Namen die Ersatzfunktion erreichen, stattdessen sind womöglich nur die jeweiligen Werte vorhanden. Diese ließen sich zwar als Liste an die Erweiterungen überge-

ben, dort wäre jedoch ein sehr unkomfortabler indexbasierter Zugriff erforderlich, falls die Parameter der Originalfunktion benötigt werden.

Um für die vorhandenen Parameterwerte die zugehörigen Namen zu finden, greift das Framework auf die in Abschnitt 7.7.5 bereits genutzte `inspect`-Bibliothek zurück. Deren Methode `getcallargs` gibt unter Angabe der Originalfunktion und der übergebenen Parameter eine Zuordnung von Namen und Werten zurück – also exakt das, was für `func_args` gesucht und somit auch genutzt wird.

### 7.7.7 Dekorierer und Python-Versionskompatibilität

Die grundlegenden Fähigkeiten und Funktionsweisen des policyextension-Frameworks sind inzwischen bekannt. Deren Implementierung jedoch muss mit einigen Problemen zurechtkommen. Diese werden im Folgenden zusammen mit ihren Auswirkungen und den dazu gefundenen Lösungen erläutert.

OpenStack macht häufig Gebrauch von Pythons Möglichkeit, unter anderem Funktionen mittels Dekorierern zu erweitern. Dazu genügt es, eine Funktion im Quellcode entsprechend zu annotieren. Ein häufiger Anwendungsfall, wenn auch nicht OpenStack-spezifisch, ist die Verarbeitung von HTTP-Anfragen durch eine Webapplikation. Diese verknüpft intern diverse URL-Routen mit Quellcode, sodass der Aufruf einer URL zur Verarbeitung durch die zugewiesene Funktion führt. Falls manche Funktionen nur `POST`-Anfragen abarbeiten sollen, ließe sich zu Beginn jeweils die Art der Anfrage überprüfen. Eleganter ist die Verwendung eines Dekorierers für diese Funktionen: Er prüft die Anfrage und wirft einen Fehler, falls es sich nicht um eine `POST`-Anfrage handelt. Ansonsten wird die eigentliche Funktion ausgeführt. Dank des Dekorierers erfolgt die Überprüfung der Anfrage mittels einer einzeiligen Annotation an den betroffenen Funktionen.

Dekorierer kapseln somit wiederverwendbaren Quellcode, der vor aber auch nach Funktionen ausgeführt werden soll. Während Dekorierer daher ausgesprochen praktisch sind, so führt ihre Art und Weise der Implementierung jedoch zu erheblichen Probleme für das policyextension-Framework. Ein Dekorierer ersetzt nämlich die annotierte Funktion durch eine andere, die er erstellt und zurückgibt. Diese neue Funktion ruft ihr Original, führt aber, je nach Dekorierer, davor oder danach eigenen Code aus.

Ein Dekorierer ist demnach ein Wrapper. Darüber hinaus ist sein Vorgehen der Methodik sehr ähnlich, wie das policyextension-Framework den Code von PolicyExtensions zur Ausführung bringt (siehe Abschnitt 7.7.4). Der Unterschied liegt in der Offensichtlichkeit der Umhüllung: Während der Einsatz des Dekorierers durch die Funktionsannotation im Quellcode ersichtlich wird, so ist das Vorgehen des Frameworks nicht zu erkennen, falls lediglich der Code der erweiterten Funktion betrachtet wird.

Dekorierer stellen eine Herausforderung für das Framework dar, wenn sie die in einer PolicyExtension referenzierte Originalfunktion erweitern. Im Parameter `func_args` erwartet der Entwickler einer Erweiterung nämlich die Argumente der Originalfunktion. Diese jedoch existiert eigentlich gar nicht mehr, da sie durch den Dekorierer ersetzt wurde. Wenn das Framework dann versucht, für die an die Originalfunktion übergebenen Parameter die zugehörigen Namen zu finden und dafür die Argumentliste der vorgeblichen Originalfunktion

inspiziert, findet es stattdessen die Argumentliste des Dekorierers vor. Dadurch kann keine Zuordnung von Werten zu Namen vorgenommen werden.

Um dennoch an die Namen zu kommen, wird eine Möglichkeit benötigt, um von der durch den Dekorierer erstellten Ersatzfunktion zur Originalfunktion zu gelangen. Mit Python 3 ist das möglich. Dekorierer nutzen intern meist selbst einen Dekorierer, nämlich `wraps`, aus dem zur Standardbibliothek gehörenden Modul `functools`. Dieser sorgt beispielsweise dafür, dass eine eventuell im Quellcode gepflegte Funktionsdokumentation bei Nutzung eines Dekorierers nicht verloren geht. Python erstellt aus dieser Dokumentation nämlich ein Funktionsattribut, das zur Laufzeit durch Tools zur automatischen Generierung von Dokumentationsunterlagen ausgelesen werden kann. Normalerweise würde die durch einen Dekorierer erstellte Ersatzfunktion kein solches Attribut zum Vorhalten der Dokumentation besitzen, sie wäre demnach für solche Tools nicht mehr verfügbar. Der `wraps`-Dekorierer hingegen übernimmt die Dokumentation der Originalfunktion und fügt sie ihrem Ersatz als Funktionsattribut hinzu. Auf dieses können die Dokumentationstools anschließend zugreifen.

In Python 3 wurde die hinter dem `wraps`-Dekorierer stehende Funktionalität erweitert, sodass die Ersatzfunktion über das Attribut `__wrapped__` die Originalfunktion referenziert. Damit ist die Zuordnung von Parameterwerten zu ihren Namen wieder möglich.

Um der noch großen Verbreitung von Python 2 Rechnung zu tragen, unterstützen viele OpenStack-Projekte beide Versionen von Python. Die Benutzung des policyextension-Frameworks soll die unterstützten Versionen nicht einschränken, sodass das Framework auch mit Python 2 funktionieren muss. Das `__wrapped__`-Funktionsattribut wird unter Python 2 allerdings nicht gesetzt. Aus diesem Grunde nimmt das Framework beim Start ein weiteres Mal Monkey Patching vor: Es übernimmt die fehlende Funktionalität aus Python 3, indem die hinter dem `wraps`-Dekorierer stehende Funktion `update_wrapper` durch eine modifizierte Variante ersetzt wird. Diese ruft zunächst ihr Original auf, fügt aber anschließend noch das `__wrapped__`-Attribut hinzu.

Die Bibliothek `six`[33] hat den Bedarf dieser Funktionalität für Python 2 erkannt und bietet einen erweiterten Dekorierer `wraps`. `six` stellt generell Hilfsmittel zur Verfügung, mit denen derselbe Quellcode sowohl unter Python 2 als auch Python 3 ausgeführt werden kann. `six.wraps` kann in eigenem Code anstelle des Dekorierers aus `functools` genutzt werden. OpenStack jedoch nutzt `functools.wraps`. Statt einer eigenen Implementierung stand auch zur Diskussion, `functools.wraps` durch `six.wraps` zu ersetzen, damit auch dekorierte OpenStack-Funktionen unter Python 2 auf ihr Original verweisen. Dies ließ sich nicht umsetzen, da `six.wraps` intern auf `functools.wraps` zurückgreift. Letztlich ist die eigene Implementierung ohnehin trivial, da lediglich ein Attribut hinzugefügt werden muss.

Die Nutzung mehrerer Dekorierer an derselben Funktion stellt kein Problem dar. Das Framework traversiert die dabei entstehende `__wrapped__`-Kette, bis es auf die Originalfunktion trifft und dort die gesuchten Parameternamen vorfindet. Da sich bekanntlich mehrere PolicyExtensions auf dieselbe Originalfunktion beziehen können, wodurch diese mehrfach umhüllt wird, nutzt auch das Framework intern den `wraps`-Dekorierer für die Ersatzfunktion, sodass auch diese einen Verweis auf das Original erhält. Für das Finden der Parameternamen

[33] `https://pythonhosted.org/six/`

ist es somit unerheblich, ob die `__wrapped__`-Kette aus Dekorierern, weiteren PolicyExtensions oder beidem besteht.

In Python werden Klassenmethoden sowie statische Methoden mittels Dekorierern umgesetzt. Zwar war keine der Originalfunktionen in den bisherigen Policy-Beispielen dekoriert, das policyextension-Framework soll diese dennoch unterstützen, um vielseitig einsetzbar zu sein. Dazu prüft es, ob es sich bei der Originalfunktion um eine statische oder eine Klassenmethode handelt und passt die Ersatzmethode entsprechend an, um einen reibungslosen Austausch zu gewährleisten.

Python 3 unterscheidet zwischen Byte-Strings (Datentyp `bytes`) und Text-Strings (Datentyp `str`). Python 2 hingegen kennt für beides nur `str`. Diese Unterschiede betreffen beispielsweise die Base64-Kodierung einer Policy, deren Ergebnis bekanntlich Binärdaten sind. Aus diesem Grunde benutzen das `formats`-Modul des Frameworks, welches PolicyFormat bereitstellt, sowie die angepasste python-keystoneclient-Bibliothek (Kapitel 5) die `six`-Bibliothek zur Sicherstellung der Versionskompatibilität.

## 7.8 Diskussion: Stärken und Schwächen

In den vorigen Abschnitten wurde das policyextension-Framework vorgestellt. Es folgt eine Diskussion über die gewählten Ansätze und inwieweit diese Stärken oder Schwächen des Framework darstellen.

**Nahezu unverändertes OpenStack** Zunächst ist festzuhalten, dass das policyextension-Framework einen Weg bietet, eine Vielzahl von OpenStack-Diensten um Unterstützung für Policies zu erweitern, dabei aber nur minimale Anpassungen am originalen Quellcode erfordert.

**PolicyExtension: Komfortables Hinzufügen der Unterstützung einer Policy** Die Implementierung einer solchen Unterstützung erfolgt mittels einer PolicyExtension. Diese haben einen klaren, immer gleichen Aufbau. Ihnen steht die auszuwertende Policy sowie die Parameterwerte der aufgerufenen Originalfunktion zur Verfügung. Damit lassen sich einfache Policies mit wenigen Zeilen Quelltext in OpenStack integrieren.

Besonders einfach ist die Erstellung einer PolicyExtension, falls diese lediglich eine Fehlermeldung ausgeben soll, wenn ein Verstoß gegen die Policy festgestellt wird. Zugleich sorgt dieser Weg dafür, dass der Verstoß nicht nur bemerkt, sondern aktiv unterbunden wird.

Alternativ kann eine PolicyExtension genauso einfach schreibend auf die für die Originalfunktion gedachten Parameterwerte zugreifen, um festgestellte Verstöße vor Auftritt zu beheben. Diese Manipulation kann allerdings Nebenwirkungen haben, deren Behandlung weit aufwändiger sein kann, als die eigentliche Umsetzung der Policy.

**Aufwand zum Finden einer Originalfunktion** Trotz der Kompaktheit von PolicyExtensions ist ihre Erstellung nicht zwingend trivial: Die Auswahl einer Originalfunktion für das Attribut

`func_paths` setzt ein zumindest grundlegendes Verständnis des originalen Quellcodes voraus. Gleiches gilt für die Nutzung der Parameter aus `func_args`.

Häufig gibt es mehrere mögliche Originalfunktionen. Diese gehören alle zur Verarbeitung derselben Anfrage, sind aber auf verschieden tiefen Ebenen der API eines Dienstes tätig. Bei der Entscheidung für eine Originalfunktion spielen verschiedene Überlegungen eine Rolle. Je tiefer die Softwareebene, desto wahrscheinlicher wurden bereits alle zur Nutzeranfrage gehörenden Eingabedaten validiert. Allerdings steigt mit zunehmender Tiefe das Risiko, dass eine Aktion bei einem Verstoß gegen die Policy nicht mehr unterbunden werden kann, sondern umständlich rückgängig gemacht werden muss.

Soll die Umsetzung der Policy schreibend auf Parameter zugreifen, gibt es einen Konflikt, da eine solche PolicyExtension hingegen auf einer möglichst hohen Ebene arbeiten sollte. Dadurch können die manipulierten Parameter durch OpenStack validiert werden und alle Überprüfungen stehen noch bevor, brauchen also nicht durch die Erweiterung mit angepassten Werten erneut ausgeführt werden.

Es bleibt festzuhalten, dass die Auswahl der Originalfunktion der womöglich anspruchsvollste Teil bei der Konstruktion einer neuen Policy-Unterstützung ist. Zugleich ist zu bedenken, dass auch bei einer Umsetzung durch Anpassungen von OpenStacks Quellcode dieselben Überlegungen getätigt werden müssten.

**CPPL ist nachrüstbar – Unabhängigkeit von der Policy-Sprache** Das Framework ist unabhängig von einer konkreten Policy-Sprache und bietet mit PolicyFormat eine Schnittstelle, über die Unterstützung für Policy-Formate und -Sprachen hinzugefügt werden kann. Auf diesem Wege kann auch CPPL Einzug in OpenStack halten.

**Monkey Patching** Es ist zunächst zu unterscheiden zwischen dem Patchen der Standardbibliothek und dem Patchen von OpenStack. Die Standardbibliothek zu modifizieren birgt ein an sich großes Risiko unbeabsichtigter Nebenwirkungen. Das policyextension-Framework patcht nur an einer Stelle das zur Standardbibliothek gehörende Modul `functools`. Diese Änderung dürfte risikofrei sein: Es wird lediglich eine simple Funktionalität aus Python 3 nach Python 2 portiert.

Mit höherem Risiko ist das Patchen von OpenStack verbunden. Ein Produktivsystem ohne dessen Wissen zu manipulieren erscheint eine nicht sonderlich gute Idee zu sein. Ganz besonders, wenn das Produktivsystem weiterentwickelt wird und mit neuen Versionen Inkompatibilitäten zu den Patches drohen. Diese Bedenken sind richtig, allerdings zielt diese Arbeit nicht auf OpenStack im produktiven Einsatz, sondern auf einen Forschungsprototypen ab. Inkompatibilitäten mit zukünftigen Versionen von Projekten, die die Originalfunktionen der PolicyExtensions enthalten, können nicht ausgeschlossen werden – dafür jedoch gibt es die Möglichkeit, die Anwendung der PolicyExtension auf spezifische Versionen zu beschränken. Des Weiteren ist dies kein Problem, das nur dem Patching zu eigen ist. Es betrifft vielmehr Plugin-Architekturen generell, bei denen Plugins (die konzeptionell den PolicyExtensions sehr ähnlich sind) von externen Entwicklern beigesteuert werden können und die nicht Teil des Quellcodes der Kernapplikation sind. Das CMS WordPress stellt aus diesem

Grunde für jedes Plugin im Plugin-Verzeichnis die Möglichkeit bereit, für eine ausgewählte WordPress- und Plugin-Version anzugeben, ob diese zueinander kompatibel sind. Die so erhobenen Daten sind öffentlich einsehbar und können vor dem Download eines Plugins bereits als Warnung dienen.

Es gibt keine Einschränkungen für die Auswahl der Originalfunktionen in den PolicyExtensions. In Kombination mit dem Patching werden die Erweiterungen dadurch sehr mächtig. Infolgedessen können neben Inkompatibilitäten viele Fehler auftreten – bis hin zu neuen Sicherheitslücken durch potentiell gezielt bösartige PolicyExtensions. Auch hier sei auf die Analogie zu WordPress-Plugins verwiesen: Als Betreiber einer OpenStack-Instanz sollten PolicyExtensions nicht blindlings hinzugefügt werden. Stattdessen handelt es sich um Code Dritter, der entsprechend kritisch beäugt und auch regelmäßig auf Aktualisierungen hin geprüft werden sollte.

Der Einsatz von Monkey Patching schließt eine Applikation nicht per se vom produktiven Einsatz aus. Laut [33] nutzen alle OpenStack-Dienste zur Nebenläufigkeit ein kooperatives Threading-Modell. Auf Nova und Cinder trifft das zu, bei Keystone findet eine Umstellung statt. Beim kooperativen Threading laufen alle erstellten Threads aus Sicht des Betriebssystems innerhalb eines einzigen Threads. Zwischen den Pseudo-Threads, *greenlets* genannt, gibt es kein implizites Scheduling. Stattdessen bestimmt die Applikation selbst, wann sie die Kontrolle an ein anderes greenlet abgibt. Ein guter Zeitpunkt ist der Aufruf von eigentlich blockierenden Operationen, beispielsweise Netzwerkzugriffe. Die zugehörigen Bibliotheken `greenlet` und `eventlet` bringen dazu bereits nicht-blockierende Operationen mit. Problematisch ist aber der Aufruf blockierender Operationen aus der Standardbibliothek, weil der gemeinsame Thread und somit alle greenlets dadurch angehalten werden würden. Aus diesem Grunde bieten `greenlet` und `eventlet` die Möglichkeit, die betroffenen Teile der Standardbibliothek durch nicht-blockierende Varianten auszutauschen. Dies geschieht ebenfalls per Monkey Patching und wird sowohl von Nova als auch Cinder genutzt. Das relativiert nicht die dem Monkey Patching inhärenten Probleme, zeigt aber, dass diese Methodik auch produktiv eingesetzt werden kann, wenn ihre Benutzung offen kommuniziert wird und ausgiebig getestet ist.

Der Austausch einer Originalfunktion durch eine Ersatzfunktion, die das Original wiederum aufruft, ist an sich wenig problematisch. Es handelt sich dabei schlicht um einen Wrapper. Dekorierer verfahren genauso. Diese haben jedoch den bedeutenden Vorteil, auf ihre Präsenz im Quellcode direkt über der Originalfunktion hinzuweisen. Die Modifikationen durch das policyextension-Framework finden stattdessen vergleichsweise versteckt statt. Bei auftretenden Fehlern kann dies die Suche nach deren Ursachen deutlich erschweren. Das Framework versucht diesen Nachteil durch aussagekräftige Fehlermeldungen abzumildern.

Nova und Cinder bringen im Übrigen bereits eine Funktionalität zum Monkey Patching mit: In ihren Konfigurationsdateien kann mittels Pfaden, ähnlich zum Format des `func_paths`-Attributs, angegeben werden, welche zur API gehörenden Funktionen mit einem weiteren Dekorierer ausgestattet werden sollen. Die zugehörigen Schlüsselwerte heißen `monkey_patch` und `monkey_patch_modules`. Auf diese Möglichkeit wurde nicht zurückgegriffen, weil sie auf Nova und das daraus hervorgegangene Cinder beschränkt ist. Während eine PolicyEx-

tension sowohl die Angabe der Originalfunktion als auch den zuvor auszuführenden Code enthält, so würde bei diesem Weg hingegen beides voneinander getrennt: Die Originalfunktion wäre in der Konfigurationsdatei referenziert, der Code läge woanders und müsste in der Konfiguration zusätzlich referenziert werden.

**Zugriff auf Policy durch Inspektion des Aufrufstapels** Den Aufrufstapel zu traversieren und nach eigentlich bereits nicht mehr verfügbaren lokalen Variablen zu suchen, verstößt im Allgemeinen gegen Prinzipien des Softwareentwurfs. Für Anwendungsfälle wie Debugger und die Ausgabe von Stacktraces ist ein solches Vorgehen hingegen essentiell.

Eine Alternative mit saubererem Design wurde bereits in Abschnitt 7.7.5 erwähnt: Eine PolicyExtension könnte gezwungen werden, nicht nur eine Originalfunktion im Attribut `func_paths` anzugeben, sondern zusätzlich einen Verweis auf weitere Funktion. In der muss die Nutzeranfrage samt Policy verfügbar ist, beispielsweise als Funktionsargument. Das Framework könnte auch diese Methode durch einen Wrapper austauschen, der neben dem Aufruf dieser zusätzlichen Originalfunktion die Policy extrahiert und sie später der PolicyExtension zur Verfügung stellt.

Dass die Entscheidung dennoch auf die Methode der Stapeldurchsuchung fiel, hat vor allem zwei Gründe: Zum einen soll das Schreiben von PolicyExtensions so einfach und kurz wie möglich gehalten werden, zum anderen ist die Methode trotz oder wegen ihrer Ungewöhnlichkeit technisch interessant, was wiederum gut zur Absicht eines Forschungsprototypen passt.

**Auffinden der Originalmethode durch Traversierung der `__wrapped__`-Kette** Um die Parameternamen der Originalfunktion zu erhalten, muss das policyextension-Framework trotz eventuell vorhandener Wrapper durch Dekorierer oder mehrere PolicyExtensions für dieselbe Originalfunktion bis zu ebendieser vordringen. Da alle Wrapper das __wrapped__-Attribut haben und somit auf die jeweils umhüllte Funktion verweisen, führt die Traversierung bis hin zur tatsächlichen Originalfunktion.

Der Ansatz schießt jedoch über sein Ziel hinaus, falls die Originalfunktion selbst ein Attribut namens __wrapped__ besitzt. Das ist jedoch hochgradig unwahrscheinlich. Funktionsattribute sind zwar möglich, scheinen aber in OpenStack nach Lektüre von Quellcode wenig gebräuchlich zu sein. Sogar Dekorierer selbst haben dieses Attribut nicht, lediglich die von ihnen zurückgegebenen Funktionen. Darüber hinaus sind Attribute mit doppelten Unterstrichen zu Beginn und zum Ende des Attributnamens für die interne Benutzung von Python selbst reserviert [36], sodass eine Einhaltung gängiger Python-Konventionen nicht zu Namenskollisionen führt.

# 8 Integration ins Web-Dashboard

OpenStack bietet neben den APIs der einzelnen Dienste und zugehörigen Kommandozeilentools auch ein webbasiertes Dashboard namens *Horizon*, über das viele Funktionalitäten graphisch verfügbar sind.

Dieses Kapitel widmet sich der Integration der bisherigen Arbeiten in Horizon. Das bedeutet, eine Oberfläche anzubieten, über die gesetzte Policies eingesehen und aktualisiert werden können sowie die Anzeige von Fehlermeldungen, die durch unterbundene Verstöße gegen Policies ausgelöst wurden. Die Integration in Horizon ist theoretisch optional, gestaltet aber die Nutzung von Policies komfortabler. Abschließend wird eine Anleitung zur Einrichtung einer Entwicklungsumgebung für Horizon gegeben.

## 8.1 Abfrage und Aktualisierung von Policies

Horizon ist ebenfalls in Python geschrieben und baut auf dem Webframework *Django*[34] auf. Horizon führt weitere Abstraktionsebenen ein, unter anderem sogenannte *Dashboards*, die registriert und daraufhin in der Weboberfläche angezeigt werden können.

Durch die Speicherung der Policies in Keystone (siehe Kapitel 4) bietet es sich an, die Funktionalität zum Abfragen und Setzen von Policies in das zu Keystone gehörende Dashboard zu integrieren. Dieses enthält eine Ansicht, in der alle Nutzer sowie einige ihrer Details tabellarisch aufgelistet werden, pro Zeile ein Nutzer. Am Ende jeder Zeile gibt es ein Menü, über das bestimmte den Nutzer betreffende Funktionen ausgeführt werden können, beispielsweise eine Passwortänderung oder die Löschung des Nutzers. Dieses Menü bietet sich eigentlich an, um eine Schaltfläche zur Änderungen der Policy hinzuzufügen.

Die weitreichenden Funktionen des Menüs sollen einfachen Nutzern, also jenen ohne administrative Rechte, allerdings nicht zur Verfügung stehen. Sowohl die Berechtigungen zur Ausführung der Funktionen als auch die Anzeige der sie startenden Schaltflächen wird über das aus Abschnitt 2.3.1 bekannte oslo.policy gesteuert. Zur Konfiguration von Keystone gehört bekanntlich die Datei `policy.json`, die festlegt, welche API-Endpunkte von wem genutzt werden dürfen. Horizon fragt diese Berechtigungen nicht von Keystone ab, sondern bringt für jeden OpenStack-Dienst eine eigene `policy.json` mit. Diese sind mit denen der Dienste möglichst synchron zu halten, andernfalls werden Nutzern in Horizon weniger Funktionen angezeigt als ihnen zustünden oder aber angezeigte Funktionen resultieren in Fehlermeldungen wegen fehlender Berechtigungen.

Die Standardwerte für `policy.json` verbieten einfachen Nutzern die Auflistung aller Nutzer. Dort aber war das infrage kommende Menü für die Policy-Bearbeitung enthalten.

[34] `https://www.djangoproject.com/`

Will man den Nutzern keine erweiterten Rechte zugestehen, muss ein anderer Platz für die Policy-Integration gefunden werden. Dafür kommt ein allen Nutzern zugängliches Menü infrage, über das bereits die Änderung des eigenen Passworts angestoßen werden kann. An dieser Stelle wurde daher eine Schaltfläche für die Policy-Funktionalität hinzugefügt, sodass jeder Nutzer die eigene Policy bearbeiten kann, falls die vergebenen Berechtigungen dies nicht unterbinden. Die Nutzerliste wurde dennoch erweitert: Sie ist für Administratoren gedacht und auch diese sollen in der Lage sein, die Policy beliebiger Nutzer anzupassen. Von daher wurde eine zusätzliche Schaltfläche zum dortigen Menü hinzugefügt, die zu den Policy-Einstellungen des jeweiligen Nutzers führt.

Es gibt keine separate Ansicht zur bloßen Anzeige einer Policy; stattdessen gibt es nur eine Bearbeitungsansicht, bei der in einem Textfeld auch die aktuelle Policy sichtbar ist. In dieser Ansicht greift Horizon auf die in Abschnitt 7.6 vorstellten Methoden eines PolicyFormats zurück, da auch Horizon unabhängig von einer konkreten Policy-Sprache sein soll. Falls ein Nutzer noch keine Policy hat, wird stattdessen das Ergebnis der Methode `get_empty_textual_policy` angezeigt. Andernfalls wird die empfangene Policy dekodiert und mit `pretty_print` in gut lesbarer Form angezeigt. Will der Nutzer eine Policy speichern, so wird zunächst `parse` aufgerufen. Falls dies zu einem Fehler der Klasse `PolicyIncompatibleFormat` führt, handelt es sich um eine invalide Policy, die der Nutzer erst korrigieren muss. Weitere Hilfestellungen zum benötigten Format erhält er nicht. Eine valide Policy wird kodiert und an Keystone gesendet. Die Kommunikation mit Keystone erfolgt über eine zu Horizon gehörende Abstraktionsschicht von python-keystoneclient (Kapitel 5).

## 8.2 Fehlerbehandlung

Horizon kommuniziert mit einer Vielzahl von OpenStack-Diensten. Dadurch kann es zu einer unüberschaubaren Menge an möglichen Fehlermeldungen kommen. Manche davon können für Nutzer relevant sein, zum Beispiel eine unerwartet nicht gestartete VM). Andere Fehlermeldungen hingegen nicht, beispielsweise wenn eine VM durch den Nutzer gelöscht wird und der zugehörige Fortschrittsbalken erst verschwindet, wenn Horizon Fehlermeldungen erhält, dass die angefragte VM nicht (mehr) existiert. Die Fehlermeldungen waren provoziert und erwartet, sie dienten Horizon als Fortschrittsindikator – einen Nutzer interessiert lediglich die erfolgreiche Löschung, nicht aber die Fehlermeldung über die nun unbekannte VM.

Aus diesem Grund versucht Horizon nicht, bei jeder Fehlermeldung eine bestimmte, an den Nutzer gerichtete Aktion auszuführen, sondern es betreibt eine zentrale Behandlung von Fehlermeldungen. Zunächst versucht Horizon, Fehler zu vermeiden, indem es teils Logik aus den OpenStack-Diensten nachimplementiert: Hat ein Nutzer sein VM-Kontingent erreicht, so würde Nova bei einer Anfrage zur Erstellung einer weiteren VM mit einem Fehler reagieren. Horizon hingegen überprüft, ob eine vom Nutzer angeforderte VM zur Überschreitung des Kontingents führen würde. Ist dies der Fall, lässt Horizon gar nicht erst zu, dass der Nutzer die Anfrage abschickt, indem es die entsprechende Schaltfläche deaktiviert. Falls dennoch Fehler auftreten, so werden einige wenige gesondert behandelt. Bei Fehlermeldungen mit

dem HTTP-Status 401 (Unauthorized) beispielsweise fordert Horizon den Nutzer auf, sich mit einem Konto mit höheren Berechtigungen anzumelden. Ein Großteil der Fehlermeldungen resultiert allerdings nur in einer nichtssagenden Benachrichtigung an den Nutzer.

Dies betrifft auch die bei Policy-Verstößen gesendeten Fehlermeldungen. Das Anlegen eines unverschlüsselten Volumes beispielsweise würde mit der Fehlermeldung *Error: Unable to create volume* fehlschlagen, falls die Policy Verschlüsselung erfordert. Um die tatsächlich von den PolicyExtensions erstellten Fehlermeldungen anzuzeigen, muss Horizon diese gesondert behandeln. Die Unterscheidung dieser Meldung von anderen ist jedoch nicht trivial, sodass sich das policyextension-Framework und Horizon eines eher unschönen Tricks bedienen: Das Framework ergänzt alle von den PolicyExtensions geworfenen Fehlermeldungen um den Präfix `PolicyViolation:`. Horizon erkennt diese Meldungen durch einen regulären Ausdruck und entfernt dabei zugleich einige vom jeweiligen Dienst angehängte Informationen, beispielsweise die ID der den Fehler auslösenden Anfrage. Die extrahierte Fehlermeldung wird schließlich dem Nutzer angezeigt.

Für Cinder gelten obige Ausführungen, allerdings ist diese zentrale Fehlerbehandlung nicht für alle Bereiche von Horizon zuständig. Einige Dashboards, wie das von Nova, implementieren eine eigene API in Horizon, die schließlich durch nutzerseitiges JavaScript angesprochen wird. Bei diesem Vorgehen wird die zentrale Fehlerbehandlung umgangen, sodass für die sich auf Nova beziehenden PolicyExtensions eine andere Lösung gefunden werden musste.

Fehlerbehandlung erfolgt im JavaScript-Frontend dezentral, das heißt, dass jeder Aufruf der Horizon-API dabei auftretende Fehler separat behandelt. Bei einer Einschränkung der Availability Zones würde ein erkannter Verstoß gegen die Policy lediglich zur Anzeige von *Error: Unable to create the server* führen. Um die aus den Policy-Beispielen aus Kapitel 7 bekannten Fehlermeldungen anzuzeigen, hätte das JavaScript an der Stelle zum Anlegen einer VM angepasst werden können. Dies wäre allerdings nur auf diesen Anwendungsfall beschränkt gewesen, benötigt wurde stattdessen eine generische Fehlerbehandlung. Aus diesem Grunde wurde der JavaScript-Code angepasst, der letztlich die HTTP-Pakete an die Horizon-API sendet. Die hinzugefügte Fehlerbehandlung analysiert die erhaltene Fehlermeldung mithilfe des oben erwähnten regulären Ausdrucks, nimmt dieselben Filterungen vor und zeigt schließlich die Fehlermeldung an.

Die Schwierigkeiten beim Anzeigen der Fehlermeldungen illustrieren den Aufwand, der mit einer kompletten Integration von Policies in OpenStack verbunden ist. Das policyextension-Framework erlaubt zwar die Erweiterung eines einzelnen Dienstes auf komfortable Weise, kann aber Nebenwirkungen in anderen Diensten nicht verhindern. Für den Umgang mit Fehlern, auch jenen, die durch Policy-Verstöße ausgelöst wurden, sind die jeweiligen Dienste zuständig. Erfahrungsgemäß handhaben sie das auch gut. Horizons Ansatz der zentralen Fehlerbehandlung ist verständlich, genügte aber den Erwartungen hinsichtlich des Nutzungserlebnisses nicht. Die Fehlerbehandlung in den von JavaScript abhängigen Dashboards bedarf allerdings einer Verbesserung.

## 8.3 Entwicklungsumgebung für Horizon

Die offizielle OpenStack-Dokumentation enthält eine vergleichsweise ausführliche Anleitung zur Einrichtung einer Entwicklungsumgebung[35]. Da anschließend aber ohnehin noch Konfigurationen vorzunehmen sind, folgt eine kurze Anleitung, die alle erforderlichen Schritte aus der Dokumentation enthält beziehungsweise diese leicht abwandelt:

- Quellcode mittels *git* herunterladen:

  ```
  $ git clone git@github.com:SSICLOPS/policy_horizon.git
  ```

- Ins geladene Verzeichnis wechseln:

  ```
  $ cd policy_horizon
  ```

- Ein virtualenv anlegen und aktivieren.
- Abhängigkeiten installieren:

  ```
  $ pip install .
  ```

- Horizon soll die angepasste Version von python-keystoneclient benutzen:

  ```
  $ pip install -e $PFAD_ZU_PYTHON_KEYSTONECLIENT
  ```

- Damit Horizon auf PolicyFormat aus dem policyextension-Framework zugreifen kann, muss es als Abhängigkeit installiert werden:

  ```
  $ pip install -e $PFAD_ZU_POLICYEXTENSION
  ```

- Horizon wird eigentlich ohne Konfigurationsdatei ausgeliefert. Der Einfachheit halber ist in der für diese Arbeit angepassten Version bereits eine fertige Konfigurationsdatei enthalten. Um sie einzusetzen, genügt ihre Umbenennung:

  ```
  $ mv openstack_dashboard/local/local_settings.py.mastersthesis \
  openstack_dashboard/local/local_settings.py
  ```

  Zu Dokumentationszwecken folgt dennoch die Erklärung, wie sie erstellt wurde.

  Beispielkonfigurationsdatei aktivieren:

  ```
  $ cp openstack_dashboard/local/local_settings.py.example \
  openstack_dashboard/local/local_settings.py
  ```

  Horizon ist auf Keystone zur Authentifizierung von Nutzern angewiesen. Dazu muss es wissen, unter welcher URL Keystone zu erreichen ist. Darüber hinaus war die Anmeldung bei Horizon nur erfolgreich, wenn die Nutzung von Keystones v3-API vorgeschrieben wurde.

  Dazu muss in obiger Konfigurationsdatei der Wert für `OPENSTACK_KEYSTONE_URL` geändert sowie ein Wert für `OPENSTACK_API_VERSIONS` hinzugefügt werden:

  ```
  OPENSTACK_API_VERSIONS = { "identity": 3, }
  OPENSTACK_KEYSTONE_URL = "http://%s:35357/v3" % OPENSTACK_HOST
  ```

[35] `http://docs.openstack.org/developer/horizon/quickstart.html`

Der in Django eingebaute Webserver liefert Horizon während der Entwicklung aus. Werden Änderungen am Quellcode festgestellt, startet der Webserver automatisch neu. In der Standardeinstellung werden dabei JavaScript- und Stylesheet-Dateien komprimiert, was den Startvorgang erheblich verzögert. Es hat sich daher als praktisch erwiesen, diese automatische Komprimierung zu deaktivieren. Dazu muss folgende Zeile zu obiger Konfiguration hinzugefügt werden:

```
COMPRESS_OFFLINE = True
```

Dies hat zur Folge, dass bei Änderungen an JavaScript- oder Stylesheet-Dateien die Komprimierung manuell ausgeführt werden muss:

```
$ python manage.py compress
```

Initial muss dieser Schritt auch ein Mal ausgeführt werden.

- Anschließend kann Horizon gestartet werden.

  ```
  $ python manage.py runserver
  ```

  Die Webapplikation steht anschließend auf Port 8000 zur Verfügung.

# 9 DevStack als Entwicklungsumgebung

Um alle in dieser Arbeit vorgenommenen Anpassungen in einem lauffähigen OpenStack auszuprobieren, bietet sich eine Installation von DevStack[36] an. Wie schon in Abschnitt 7.5.4 beschrieben, handelt es sich dabei nicht um ein produktiv einzusetzendes OpenStack, sondern eine für Entwickler gedachte Installation.

Die Einrichtung von DevStack erfolgt durch Ausführung eines einzelnen Skripts. Empfehlenswert ist zuvor das Anlegen einer Konfigurationsdatei um zu bestimmen, welche Dienste installiert werden sollen. Da bei der Installation diverse Netzwerkeinstellungen geändert werden, ist die Installation innerhalb einer VM ratsam. Damit darin wiederum die von DevStack erstellten VMs ausgeführt werden können, muss eine Kombination aus Hardware und Virtualisierungslösung zum Einsatz kommen, die *Nested Virtualization* unterstützt, softwareseitig beispielsweise *libvirt* mit *kvm/qemu*[37].

Um die vorgenommenen Änderungen in DevStack zu integrieren, wurde auch dieses leicht angepasst[38], damit es beispielsweise Cinder und Nova zur Nutzung der policymiddleware konfiguriert.

In der einfachsten Variante laufen alle durch DevStack installierten Services innerhalb der DevStack-VM. Um das Policy-Beispiel zur Einschränkung von Availability Zones (Abschnitt 7.4) zu veranschaulichen, bietet sich eine OpenStack-Instanz mit mindestens zwei Availability Zones an. Dazu wiederum bedarf es mindestens zweier (virtueller) Hosts, auf denen mit Nova gestartete VMs ausgeführt werden können. Dazu muss die DevStack-Installation auf mehrere Maschinen verteilt werden. Dies ist mit DevStack möglich und benötigt Anpassungen an der Konfiguration.[39]

Um den Aufwand zur Einrichtung eines verteilten DevStacks mit den Anpassungen dieser Arbeit zu erleichtern, werden im Folgenden zwei fertige auf Ubuntu 14.04 basierende VMs und ihre Installation vorgestellt.

- Die erste VM, genannt *main* ist ein normales DevStack. Die zweite DevStack-VM, genannt *compute*, ist auf die Ausführung von über Nova gestartete VMs reduziert und verbindet sich dazu mit main. Die zu den VMs gehörenden Abbilder sind zur Ausführung mit *libvirt* und *kvm/qemu* gedacht. Sie stehen unter XXX zur Verfügung.
- Neben den eigentlichen Abbildern benötigt libvirt Beschreibungen zum einen der virtuellen Hardware der VMs und zum anderen der virtuellen Netzwerke, an die die

[36] `http://docs.openstack.org/developer/devstack/`
[37] `https://libvirt.org/`
[38] `https://github.com/SSICLOPS/policy_devstack`
[39] `http://docs.openstack.org/developer/devstack/guides/multinode-lab.html`

VMs angeschlossen werden sollen. Die Beschreibungen liegen als XML-Dateien vor und gibbet hier XXX. Links

- Diese müssen zunächst angepasst werden, um den korrekten Pfad zum jeweiligen Abbild zu enthalten. Dazu ist jeweils Zeile 28 der Dateien `main.xml` und `compute.xml` anzupassen.
- Anschließend müssen die von den VMs erwarteten Netzwerke in libvirt erstellt werden. Dazu genügt der Import ihrer XML-Beschreibungen und die anschließende Aktivierung:

```
$ virsh net-define net_devstack_hosts.xml
$ virsh net-define net_devstack_floating.xml
$ virsh net-start devstack_hosts
$ virsh net-start devstack_floating
```

- Schließlich können beide VMs importiert und gestartet werden:

```
$ virsh define main.xml
$ virsh define compute.xml
$ virsh start main
$ virsh start compute
```

- Ein durch DevStack bereitgestelltes OpenStack übersteht nicht den Neustart der DevStack-VM, sodass OpenStack in den importieren VMs manuell gestartet werden muss. Der Zugriff auf die VMs erfolgt über SSH mit dem Nutzernamen `vagrant` und dem Passwort `vagrant`. Die main-VM ist unter der IP-Adresse `192.168.27.100` verfügbar, die compute-VM unter `192.168.27.101`.
- Beide VMs haben im Nutzerverzeichnis von `vagrant` den Ordner `devstack`. Darin befindet sich jeweils das Skript `stack.sh`, welches OpenStack startet, dafür allerdings mehrere Minuten in Anspruch nimmt. Dies ist zuerst auf der main-VM durchzuführen, nach Abschluss ebenfalls auf der compute-VM.
- Nachdem auch dort das Skript beendet ist, müssen Availability Zones angelegt werden. Durch die Verbindung beider VMs ist es unerheblich, in welcher VM die folgenden Anweisungen ausgeführt werden.

```
$ export OS_USERNAME=admin
$ nova aggregate-create agg1 az1
$ nova aggregate-create agg2 az2
$ nova aggregate-add-host agg1 main
$ nova aggregate-add-host agg2 compute
```

  Danach steht die main-VM in der Availability Zone `az1`, die compute-VM in `az2` zur Verfügung.
- Um verschlüsselte Volumes mit Cinder anlegen zu können, sind die in Abschnitt 7.3.2 angegebenen Kommandos zur Erstellung eines VolumeTypes auszuführen.
- Schließlich sind sowohl alle APIs als auch das Horizon-Dashboard unter der IP-Adresse `192.168.27.100` erreichbar. Die Authentifizierung kann über die Nutzernamen `admin` und `demo` erfolgen, für beide lautet das Passwort `password`.

- Basierend auf den in Kapitel 7 vorgestellten Beispielen könnte eine Policy wie folgt angegeben werden:

```
{
    "images": {
        "replication": [
            {
                "rate": 3,
                "name": "cirros-0.3.4-x86_64-uec"
            }
        ]
    },
    "availability_zones": [
        "az2"
    ],
    "storage": {
        "encryption": true
    }
}
```

Wenn DevStack erst einmal funktioniert, ist es sehr nützlich für Entwickler. Bis dahin und auch danach kann es jedoch zu Problemen kommen. Daher folgen einige Ratschläge und Tipps:

- Falls nicht die bereitgestellten VMs genutzt werden und dort das angepasste DevStack installiert werden soll, wird es vermutlich zu Problemen kommen. Der Grund ist, dass zwar die OpenStack-Projekte in der Mitaka-Version installiert werden, für die Python-Abhängigkeiten jedoch keine fixen Versionen verlangt werden. Häufig gibt es zwar Einschränkungen, jedoch nicht immer. Die Python-Abhängigkeiten führen daher in Zukunft womöglich zu Inkompatibilitäten.

  Ein ähnliches Problem trat in den bereitgestellten VMs auf: Nach erstmaligem Fehlschlag von `stack.sh` ließ sich das Problem umgehen, indem in der Datei `/opt/stack/requirements/upper-constraints.txt` die zu installierende Version von `python-openstacksdk` auf `0.9.11` gesetzt wird. Der Fehler ist bekannt[40], aber noch nicht zufriedenstellend gelöst.
- Der Quellcode der Dienste und der angepassten Bibliotheken ist im Verzeichnis `/opt/stack` zu finden.
- Aktualisierungen lassen sich dort in den Unterverzeichnissen gezielt mittels `git pull` durchführen.
- Einmal geladener Quellcode wird bei erneuter Ausführung von `stack.sh` wiederverwendet. Ein erneutes Laden lässt sich über die Umgebungsvariable `RECLONE=yes` erzwingen.
- Bevor `stack.sh` erneut ausgeführt werden kann, muss zunächst `unstack.sh` aufgerufen werden.

[40] `https://bugs.launchpad.net/python-openstacksdk/+bug/1588823`

- Die Log-Dateien werden unter `/opt/stack/logs` gespeichert, sind aber optisch ansprechender über `screen -r stack` abrufbar.

# 10 Zusammenfassung und Ausblick

Der zunehmende Einsatz von Cloud-Computing wirft Fragen des Datenschutzes auf. Bislang müssen Nutzer sich mit den pauschalen Zusicherungen der Anbieter von Cloud-Infrastrukturen zufrieden geben. Das SSICLOPS-Projekt arbeitet am Ansatz, dieses Prinzip umzukehren: Nutzer spezifizieren ihre Anforderungen und die Cloud-Anbieter müssen sich daran halten. Diese Anforderungen werden mittels Policies ausgedrückt.

Mit CPPL ist im SSICLOPS-Projekt eine Policy-Sprache entwickelt worden, die für den Einsatz in Cloud-Umgebungen optimiert ist. Bisherige Clouds unterstützten keine von Nutzern vorgegebenen Policies. Aus diesem Grunde soll OpenStack, ein Projekt zur Entwicklung freier Software zum Betrieb einer Cloud nach dem IaaS-Modell, dahingehend erweitert werden.

Policies können für eine Vielzahl von Anwendungsfällen formuliert werden. Der einschränkende Faktor sind nur die Cloud-Plattformen, die um Unterstützung für die Policies erweitert werden müssen. Die Vielzahl möglicher Policies bedeutet einen enormen Entwicklungsaufwand. Während die bloße nachträgliche Feststellung von Verstößen gegen Policies vergleichsweise einfach umzusetzen ist, so ist ihre proaktive Verhinderung, sodass es überhaupt nicht erst zu Verstößen kommt, deutlich aufwändiger. Zur Feststellung eines Verstoßes genügt eine Komponente, die Informationen über den Zustand der Cloud auswertet und mit den Policy-Vorgaben vergleicht. Die Verhinderung hingegen erfordert, dass die einzelnen Komponenten einer Cloud die Policy aktiv berücksichtigen. Dies erfordert die Anpassung all dieser Komponenten. Der damit verbundene Aufwand für Implementierung und die zugehörige Koordinierung der Entwickler währenddessen stellen ein wesentliches Hemmnis zur Umsetzung des geplanten Ansatzes dar.

Diese Arbeit hat einen möglichen Weg aufgezeigt, wie Policies dennoch in OpenStack implementiert werden können. Dabei ging es vor allem um die Schaffung der Infrastruktur, mit der die Unterstützung für einzelne Policies ermöglicht werden kann. Alle Entwurfsentscheidungen basierten auf dem Ziel, die Implementierung einer neuen Policy-Unterstützung so einfach wie möglich zu halten. Dies soll das erwähnte Umsetzungshemmnis verringern und somit die Wahrscheinlichkeit eines tatsächlichen Einsatzes erhöhen.

Da Policies durch Nutzer vorgegeben werden, erfolgt ihre Speicherung als Teil des Datenmodells eines Nutzers in der OpenStack-Komponente zur Nutzerverwaltung, Keystone. Um die Policy von Keystone abfragen und auch aktualisieren zu können, wurden Keystones API sowie die zugehörige Client-Bibliothek entsprechend erweitert.

CPPL sieht vor, dass jede Anfrage eines Nutzers mit einer Policy versehen sein kann. Dafür wurde eine Middleware entwickelt, die Nutzeranfragen um die Policy der jeweiligen Nutzer ergänzt. Aus Sicht der OpenStack-Dienste ist die Policy somit bereits Teil der eingehenden Anfrage und sie müssen sich nicht selbst um deren Beschaffung kümmern.

Der Kern dieser Arbeit lag im Entwurf und der Implementierung des policyextension-Frameworks. Es ermöglicht die einfache Erweiterung eines OpenStack-Dienstes um Unterstützung für Policies. Eine konkrete Policy erfordert dabei die Implementierung einer PolicyExtension. Deren Quellcode wird durch das Framework zur Laufzeit in OpenStack eingebunden. Eine PolicyExtension referenziert dazu eine Funktion innerhalb des OpenStack-Quellcodes. Der Erweiterungscode wird unmittelbar vor jenem der Originalfunktion ausgeführt. PolicyExtensions sind nicht Teil des offiziellen OpenStack-Quellcodes, sondern sind eher vergleichbar mit Plugins, die auch durch Dritte erstellt werden können. Das hat zur Folge, dass der Quellcode eines OpenStack-Dienstes nur minimal angepasst werden muss, um vorhandene PolicyExtensions zu unterstützen: Es genügt ein Funktionsaufruf, der beim Start die PolicyExtensions aus einem angegebenen Verzeichnis lädt.

PolicyExtensions erhalten vom Framework die gültige Nutzer-Policy sowie die an die Originalfunktion übergebenen Parameter. Eine PolicyExtension kann sich daher auf die Auswertung der Policy-Vorgaben konzentrieren und braucht bei Feststellung, dass die Anfrage zu einem Verstoß gegen die Policy führen würde, lediglich einen Fehler zu werfen. Alternativ kann die PolicyExtension auch schreibend auf die Parameter der Originalfunktion zugreifen und somit einen bevorstehenden Verstoß verhindern. Insgesamt ist es somit möglich, die Unterstützung einer neuen Policy in OpenStack sehr kompakt durch eine PolicyExtension zu bewerkstelligen. Der womöglich schwierigste Teil bei der Erstellung einer neuen PolicyExtension ist das Finden einer geeigneten Originalfunktion, weil dazu zumindest der Quellcode der oberen Software-Ebenen der API des jeweiligen OpenStack-Dienstes betrachtet werden muss.

Zur Veranschaulichung wurden mithilfe des policyextension-Frameworks drei Policies beispielhaft umgesetzt:

1. OpenStack muss die virtuellen Festplatten eines Nutzers serverseitig verschlüsseln.
2. Nutzer schränken die Verwendung von Cloud-Ressourcen auf bestimmte geographische Gebiete ein.
3. Nutzer verlangen die replizierte Ausführung bestimmter VMs.

Trotz der Fokussierung von CPPL auf den Einsatz in Cloud-Umgebungen ist das policyextension-Framework unabhängig von einer konkreten Policy-Sprache. Für diese Arbeit wurde zunächst eine JSON-Syntax gewählt. Es stehen jedoch Schnittstellen bereit, um das Framework um Unterstützung für CPPL zu ergänzen.

Abschließend wurde OpenStacks Weboberfläche erweitert, sodass Nutzer dort ihre Policies hinterlegen und bearbeiten können. Führt die Anfrage eines Nutzers durch einen abgewendeten Policy-Verstoß zu einer Fehlermeldung, so wird er darüber informiert, unabhängig davon, ob er die Weboberfläche, die Kommandozeilentools der einzelnen OpenStack-Dienste oder deren APIs benutzt.

Die erarbeiteten Konzepte und die zugehörige Software haben sich als funktional erwiesen. Nichtsdestotrotz gibt es bereits eine Reihe von Verbesserungsmöglichkeiten und weitere Ideen, die im Rahmen einer Weiterentwicklung betrachtet werden sollten. Diese werden im Folgenden jeweils kurz erläutert.

**Detektion unterstützter Policies** Der bisherigen Version fehlt noch ein Mechanismus zur Detektion, welche Policies in welchem Format von der Cloud unterstützt werden. Diese Informationen benötigt der Nutzer zur Formulierung seiner Vorgaben. Denkbar wäre, dass das policyextension-Framework von einer PolicyExtension erwartet, dass sie diese Informationen bereitstellt. Über einen vom Framework angelegten API-Endpunkt könnten diese Informationen bezogen werden. Der Endpunkte wäre jedoch pro Dienst vorhanden. Eine zentrale Stelle, wie sie das Congress-Projekt (siehe Abschnitt 2.3.5) bieten könnte, wäre für Nutzer komfortabler. Generell wäre es einfacher, wenn ein solches Feature nicht durch Monkey Patching bereitgestellt werden muss, sondern die OpenStack-Dienste sich der Existenz von Policies bewusst wären und von sich aus einen API-Endpunkt anbieten würden.

**Policies auf Gruppenebene** Policies werden von Nutzern vorgegeben. Denkbar wäre auch, dass eine Firma ihren Mitarbeitern Policies zur Nutzung von Cloud-Ressourcen vorschreiben will. In einem solchen Szenario wäre es hilfreich, wenn Policies nicht nur auf einen Nutzer bezogen wären, sondern es auch Policies für Gruppen gäbe. Keystone kennt bereits das Konzept von Gruppen, die Policy-Implementierung müsste aber entsprechend erweitert werden. Denkbar wäre beispielsweise, dass bei einem Konflikt zwischen Gruppen- und Nutzer-Policy jene der Gruppe Vorrang hat.

**Schreibende Policy zur Festplattenverschlüsselung** Von den drei Beispiel-Policies greift nur die letzte zur Replikation von VMs schreibend auf die Parameter der Originalfunktion zu. Der Großteil der Implementierung der zugehörigen PolicyExtension behandelt einige dadurch womöglich entstehende Nebenwirkungen. Das erste Policy-Beispiel zur Festplattenverschlüsselung könnte besser für eine Manipulation der Anfrage geeignet sein: Verlangt die Policy Verschlüsselung, ohne dass ein Nutzer dies explizit in der Anfrage zur Erstellung einer virtuellen Festplatte mitteilt, könnte, falls nötig, zunächst ein VolumeType zur Verschlüsselung konfiguriert und dieser dann ausgewählt werden. Da die Verschlüsselung transparent auf Serverseite erfolgt, bekommt der Nutzer davon ohnehin nichts mit, sodass ihm die automatisch zugeschaltete Verschlüsselung keine Probleme bereiten sollte.

**Versionsrestriktionen auf mehrere Pakete ausdehnen** Der Monkey Patching-Ansatz, der zur Integration der PolicyExtensions in eine laufende OpenStack-Instanz genutzt wurde, hat den Nachteil, dass Inkompatibilitäten bei einer Aktualisierung auf eine neuere OpenStack-Version auftreten können. Um dies zu vermeiden, können PolicyExtensions ihre Ausführung auf bestimmte Versionen eines Python-Paketes beschränken. Eine zukünftige Version des policyextension-Frameworks sollte einer PolicyExtension erlauben, Versionsrestriktionen für mehrere Pakete anzugeben. Da OpenStack eine Vielzahl von Abhängigkeiten hat, könnte auch eine PolicyExtension diese nutzen, aber sich selbst auf eine genauer spezifizierte Umgebung beschränken.

**Code nach der Originalfunktion ausführen** Eine PolicyExtension wird momentan immer unmittelbar vor der von ihr angegebenen Originalfunktion ausgeführt. Das policyextension-

Framework könnte erweitert werden, um auch die Ausführung danach zu erlauben. Eine Ausführung nach der Originalfunktion könnte beispielsweise die Behandlung möglicher Nebenwirkungen beim dritten Policy-Beispiel zur VM-Replikation vereinfachen: Die PolicyExtension könnte sich auf einer höheren Ebene einklinken und zunächst wieder vor der Originalfunktion laufen. Dabei würde sie die Anzahl angefragter VMs wenn nötig erhöhen. Die bisher in der Erweiterung vorhandenen und duplizierten Validierungen würden wegfallen, weil OpenStack diese durch das frühere Einklingen selbst ausführen würde. Falls die erhöhte VM-Anzahl bei der Validierung zu Fehlern führt, so könnten diese im Teil, der nach der Originalfunktion ausgeführt wird, abgefangen und stattdessen die bekannte Inkompatibilitätsfehlermeldung ausgegeben werden.

**policyextension-Framework könnte Middleware womöglich ersetzen** Die Middleware ist momentan dafür verantwortlich, eine eingehende Anfrage um die zugehörige Policy zu ergänzen. Aus Sicht eines OpenStack-Dienstes ist dies praktisch, weil dieser sich zumindest um diesen Aspekt nicht mehr kümmern muss. Dank des policyextension-Frameworks muss sich der Dienst aber ohnehin um überhaupt nichts kümmern, außer initial die Erweiterungen zu laden. Insofern könnte auch das Framework für die Policy-Abfrage zuständig sein. Für den eigentlichen Dienst bliebe es genauso einfach, jedoch hätte dieser Weg weniger unnütze Anfragen bei Keystone zur Folge: Momentan führt jede Anfrage bei einer API eines Dienstes, der um Policy-Unterstützung erweitert wurde, zu einer Abfrage der Policy bei Keystone – unabhängig davon, ob der konkrete API-Endpunkt von einer Policy betroffen ist. Darüber hinaus wäre es nicht länger nötig, die Policy durch Inspektion des Aufrufstapels zu ermitteln. Für eine zukünftige Entwicklung ist daher zu evaluieren, ob die Middleware noch benötigt wird.

**Zwischenlösungen auf dem Weg zu nativer Policy-Unterstützung in OpenStack** Monkey Patching hat sich als geeignet erwiesen, um in kurzer Zeit Erfolge bei der Integration von Policies in OpenStack zu erzielen. Dies gilt besonders, wenn die Policy zu einer Restriktion einer bereits vorhandenen Funktionalität in OpenStack führt. Die Implementierung gänzlicher neuer Funktionalitäten ist zwar auch mittels Monkey Patching möglich, ist jedoch mit mehr Aufwand verbunden und wäre in OpenStack selbst besser aufgehoben. Langfristig muss das Ziel daher sein, dass OpenStack Unterstützung für Policies mitbringt. Ein Zwischenziel könnte sein, die Originalfunktion mit einem zusätzlichen Dekorierer zu annotieren. Dieser könnte für die Einbindung der PolicyExtensions zuständig sein. Auf diesem Wege wäre auch im OpenStack-Quellcode selbst ersichtlich, dass zusätzliche Logik benutzt wird.

Eine PolicyExtension würde in einem solchen Szenario nicht zwingend selbst die Policy auswerten und darauf gegebenenfalls reagieren, sondern könnte stattdessen Kontakt zu einem separaten Dienst zur Durchsetzung von Policies, beispielsweise Congress, aufnehmen und von diesem die Entscheidung einholen, wie mit der aktuellen Anfrage zu verfahren ist. Dieser Weg wäre zugleich ein Ansatz, um das grundlegende Problem von Congress zu lösen: Policy-Verstöße lassen sich nachträglich feststellen, aber die Unterbindung vor ihrem Auftritt

erfordert die Kooperation der jeweiligen Dienste. Diese Aufgabe könnten die PolicyExtensions übernehmen.

# Literaturverzeichnis


[1] Kersten Auel. *Azure Deutschland: Microsofts Cloud-Dienste für Geschäftskunden in Europa*. Abgerufen am: 17.11.2016. Sep. 2016. URL: https://heise.de/-3328908.

[2] Oren Ben-Kiki, Clark Evans und Brian Ingerson. *YAML Ain't Markup Language (YAML™) Version 1.1*. Technischer Bericht. yaml.org, Tech. Rep, 2005.

[3] Tim Berners-Lee, Roy T. Fielding und Henrik Frystyk Nielsen. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945. RFC Editor, Mai 1996.

[4] Christian Bettstetter und Christoph Renner. „A comparison of service discovery protocols and implementation of the service location protocol". In: *Proceedings of the 6th EUNICE Open European Summer School: Innovative Internet Applications*. 2000.

[5] Douglas Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627. RFC Editor, Juli 2006.

[6] Deutsche Telekom. *Einfach. Sicher. Günstig: Die Public-Cloud-Alternative aus Deutschland*. Abgerufen am: 17.11.2016. URL: https://cloud.telekom.de/de/infrastruktur/open-telekom-cloud/.

[7] Felix Eberhardt, Jens Hiller, Oliver Hohlfeld, Stefan Klauck, Max Plauth, Andreas Polze, Matthias Uflacker und Klaus Wehrle. *Design of Inter-Cloud Security Policies, Architecture, and Annotations for Data Storage*. Technischer Bericht. Jan. 2016.

[8] Felix Eberhardt, Max Plauth, Andreas Polze, Stefan Klauck, Matthias Uflacker, Jens Hiller, Oliver Hohlfeld und Klaus Wehrle. *D2.1: Report on Body of Knowledge in Secure Cloud Data Storage*. Technischer Bericht. Juni 2015.

[9] Phillip J. Eby. *PEP 333 – Python Web Server Gateway Interface v1.0*. Python Software Foundation, Dez. 2010. URL: https://www.python.org/dev/peps/pep-0333/.

[10] Moritz Förster. *AWS Summit: Sicherheit und Compliance für die Cloud*. Abgerufen am: 17.11.2016. Juli 2016. URL: https://heise.de/-3254229.

[11] Moritz Förster. *Cloud Computing: Dropbox speichert in Deutschland*. Abgerufen am: 17.11.2016. Sep. 2016. URL: https://heise.de/-3330939.

[12] Craig Gentry. „A fully homomorphic encryption scheme". Dissertation. Stanford University, 2009.

[13] Simon Godik, Anne Anderson, Bill Parducci, P Humenn und S Vajjhala. *OASIS eXtensible Access Control Markup language (XACML)*. Technischer Bericht. Tech. rep., OASIS, Mai 2002.

[14] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux und Samuel Madden. „HYRISE: A Main Memory Hybrid Storage Engine“. In: *Proc. VLDB Endow.* 4.2 (Nov. 2010), Seiten 105–116. ISSN: 2150-8097. DOI: 10.14778/1921071.1921077. URL: http://dx.doi.org/10.14778/1921071.1921077.

[15] Martin Holland. *Umfrage: Deutsche Firmen scheuen "Cloud" auch wegen "Bauchgefühls"*. Abgerufen am: 17.11.2016. Okt. 2016. URL: https://heise.de/-3358260.

[16] Khanh-Toan Tran and Jérôme Gallard. *A new mechanism for nova-scheduler : Policy-based Scheduling*. Abgerufen am: 22.11.2016. 2013. URL: https://docs.google.com/document/d/1gr4Pb1ErXymxN9QXR4G_jVjLqNOg2ij9oA0JrLwMVRA/edit.

[17] KPMG. *Cloud-Monitor 2016: Die Mehrheit nutzt die Wolke*. Abgerufen am: 17.11.2016. Mai 2016. URL: https://home.kpmg.com/de/de/home/themen/2016/05/cloud-monitor-2016.html.

[18] Fabian Maschler, Jan-Henrich Mattfeld und Norman Rzepka. „Scalable and Secure Infrastructures for Cloud Operations“. In: *Proceedings of the Fourth HPI Cloud Symposium "Operating the Cloud" 2016*. to appear. Potsdam, Germany, 2016.

[19] Peter Mell und Timothy Grance. *The NIST Definition of Cloud Computing*. Technischer Bericht 800-145. Gaithersburg, MD: National Institute of Standards und Technology (NIST), Sep. 2011. URL: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[20] OpenStack Foundation. *Blueprints*. Abgerufen am: 24.11.2016. URL: https://wiki.openstack.org/wiki/Blueprints.

[21] OpenStack Foundation. *Congress*. Abgerufen am: 23.11.2016. URL: https://www.openstack.org/software/releases/mitaka/components/congress.

[22] OpenStack Foundation. *Congress User Stories/Use Cases*. Abgerufen am: 22.11.2016. URL: https://docs.google.com/document/d/1ExDmT06vDZjzOPePYBqojMRfXodvsk0R8nRkX-zrkSw/edit.

[23] OpenStack Foundation. *Design Summit*. Abgerufen am: 24.11.2016. URL: https://wiki.openstack.org/wiki/Design_Summit.

[24] OpenStack Foundation. *Developer's Guide*. Abgerufen am: 24.11.2016. URL: http://docs.openstack.org/infra/manual/developers.html.

[25] OpenStack Foundation. *Devstack*. Abgerufen am: 29.11.2016. URL: http://docs.openstack.org/developer/devstack/.

[26] OpenStack Foundation. *Filter Scheduler*. Abgerufen am: 22.11.2016. URL: http://docs.openstack.org/developer/nova/filter_scheduler.html.

[27] OpenStack Foundation. *GroupBasedPolicy*. Abgerufen am: 22.11.2016. URL: https://wiki.openstack.org/wiki/GroupBasedPolicy.

[28] OpenStack Foundation. *OpenStack Services*. Abgerufen am: 18.11.2016. URL: https://www.openstack.org/software/project-navigator.

[29] OpenStack Foundation. *OpenStack User Stories*. Abgerufen am: 18.11.2016. URL: https://www.openstack.org/user-stories/.

[30] OpenStack Foundation. *Policy as a service ("Congress")*. Abgerufen am: 22.11.2016. URL: https://wiki.openstack.org/wiki/Congress.

[31] OpenStack Foundation. *Scaling*. Abgerufen am: 14.12.2016. URL: http://docs.openstack.org/ops-guide/arch-scaling.html#segregating-your-cloud.

[32] OpenStack Foundation. *Technical considerations*. Abgerufen am: 15.12.2016. URL: http://docs.openstack.org/arch-design/massively-scalable-technical-considerations.html#segregation-example.

[33] OpenStack Foundation. *Threading model*. Abgerufen am: 24.01.2017. URL: http://docs.openstack.org/developer/nova/threading.html.

[34] OpenStack Foundation. *VolumeEncryption*. Abgerufen am: 06.12.2016. URL: https://wiki.openstack.org/wiki/VolumeEncryption.

[35] OpenStack Foundation. *Welcome to Congress!* Abgerufen am: 22.11.2016. URL: http://docs.openstack.org/developer/congress/.

[36] Guido van Rossum, Barry Warsaw und Nick Coghlan. *PEP 8 – Style Guide for Python Code*. Python Software Foundation, Juli 2001. URL: https://www.python.org/dev/peps/pep-0008/.

[37] Maxim Schnjakin und Christoph Meinel. „Implementation of Cloud-RAID: A Secure and Reliable Storage above the Clouds“. In: *Grid and Pervasive Computing: 8th International Conference, GPC 2013 and Colocated Workshops, Seoul, Korea, May 9-11, 2013. Proceedings*. Herausgegeben von James J. (Jong Hyuk) Park, Hamid R. Arabnia, Cheonshik Kim, Weisong Shi und Joon-Min Gil. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, Seiten 91–102. ISBN: 978-3-642-38027-3. DOI: 10.1007/978-3-642-38027-3_10. URL: http://dx.doi.org/10.1007/978-3-642-38027-3_10.

[38] David Schwalb, Jan Kossmann, Martin Faust, Stefan Klauck, Matthias Uflacker und Hasso Plattner. „Hyrise-R: Scale-out and Hot-Standby Through Lazy Master Replication for Enterprise Applications“. In: *Proceedings of the 3rd VLDB Workshop on In-Memory Data Management and Analytics*. IMDM '15. Kohala Coast, HI, USA: ACM, 2015, 7:1–7:7. ISBN: 978-1-4503-3713-7. DOI: 10.1145/2803140.2803147. URL: http://doi.acm.org/10.1145/2803140.2803147.

[39] Daniel AJ Sokolov. *US-Regierung kämpft weiter um Zugriff auf EU-Rechenzentren*. Abgerufen am: 17.11.2016. Okt. 2016. URL: https://heise.de/-3352461.

[40] Synergy Research Group. *Amazon Leads; Microsoft, IBM & Google Chase; Others Trail*. Abgerufen am: 18.11.2016. Aug. 2016. URL: https://www.srgresearch.com/articles/amazon-leads-microsoft-ibm-google-chase-others-trail.

[41] Tilman Wittenhorst. *BSI veröffentlicht Anforderungskatalog für Cloud Computing*. Abgerufen am: 17.11.2016. März 2016. URL: https://heise.de/-3141368.