

Middleware – Cloud Computing – Übung

Hybride Cloud: OpenStack - Private Cloud

Wintersemester 2020/21

Michael Eischer, Laura Lawniczak, Tobias Distler

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)

www4.cs.fau.de



Lehrstuhl für Verteilte Systeme
und Betriebssysteme



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

OpenStack

OpenStack

- Web-Frontend
 - Dashboard: <https://i4cloud1.cs.fau.de>
 - Zugangsdaten: siehe Gruppeneinteilungs-E-Mail
- Kommandozeilen-Client
 - OpenStack-Client-Programm: `openstack`
 - **Vor Verwendung:** `openrc`-Datei `sourcen` (siehe unten)
- Alle Kommandozeilenbefehle benötigen vorherige Authentifizierung
 - 1) Download der RC-Datei (`<user>-openrc.sh`) über Dashboard:
 - „Projekt“ → „API Access“
 - „Download OpenStack RC File“
 - 2) RC-Datei einlesen und ausführen (`sourcen`)

```
$ source /path/to/<user>-openrc.sh
```

- Benutzerdaten für Login per OpenStack-Konsole auf einer laufenden Instanz des bereitgestellten Beispielabbilds (`debian-example`):
USER: `cloud` PASSWORD: `cloud`

- OpenStack4j: Java-API für OpenStack-Dienste
 - Bibliotheken: /proj/i4mw/pub/aufgabe2/openstack4j-3.3
 - Dokumentation: <https://openstack4j.github.io/learn>

■ Authentifizierung

```
// Package: org.openstack4j.model.common
Identifier userDomainName = Identifier.byName(<user_domain_name>);
Identifier projectIdentifier = Identifier.byId(<project_identifizier>);
OSClientV3 client = OSFactory.builderV3() // Packages:
    .endpoint(<os_auth_url>) // org.openstack4j.{api,openstack}
    .credentials(<user>, <pass>, userDomainName)
    .scopeToProject(projectIdentifier)
    .authenticate();
```

- Parameter in OpenStack RC-Datei
 - Benutzer-Domänen-Name (<user_domain_name>): Variable OS_USER_DOMAIN_NAME
 - Projekt-ID (<project_identifizier>): Variable OS_PROJECT_ID
 - Endpunkt-Adresse (<os_auth_url>): Variable OS_AUTH_URL
- Benutzername (<user>) und Passwort (<pass>): siehe E-Mail zur Gruppeneinteilung
- OSClientV3 ist an Thread gebunden → Neuen Client für anderen Thread per OSFactory.clientFromToken(client.getToken()) erzeugen

■ Konfiguration (ähnlich zu AWS-API) über ServerCreate-Objekt

```
ServerCreate sc = Builders.server() // org.openstack4j.model.compute.api
    .<config_option1>
    .<config_option2>[...].<config_optionN>.build();
```

- Konfigurieren von Instanzname, Instanztyp (Flavor-**ID**), Abbild-**ID**, Keypair, Netzwerk-**ID**, Security-Group, UserData (Kodierung mittels `java.util.Base64`)
- Ersteinrichtung: Siehe Übung zum „Erstellen eines VM-Abbilds für OpenStack“

■ Boot mit Konfiguration (Aufruf blockiert, bis VM aktiv ist)

```
Server server = client.compute().servers()
    .bootAndWaitActive(sc, <max_wait_time_in_ms>);
```

■ Statusabfrage

`org.openstack4j.model.compute.Server.Status`

```
String serverId = server.getId();
Status st = client.compute().servers().get(serverId).getStatus();
```

- VM hat initial nur interne IP

→ Zugriff von extern nur mit Floating-IP möglich

- Floating-IP zuweisen

org.openstack4j.model.{compute,common}

```
List<? extends FloatingIP> ips = client.compute().floatingIps().list();
FloatingIP floatingIp = ips.get(0);
// [...] unbenutzte IP mit (floatingIp.getInstanceId() == null) suchen
ActionResponse r = client.compute().floatingIps().addFloatingIP(server,
    floatingIp.getFloatingIpAddress());
```

- Floating-IP abfragen

org.openstack4j.model.{compute,common}

```
String publicIp = "";
List<? extends Address> vmAddresses = server.getAddresses().getAddresses("internal");
for (Address address: vmAddresses) {
    if (address.getType().equals("floating") && address.getVersion() == 4) {
        publicIp = address.getAddr();
        break;
    }
}
```

Zugriff auf Metriken in OpenStack mittels Gnocchi

- Datenabruf per REST-Anfragen
 - Zugriff über `WebTarget`-Objekt
 - Dokumentation: https://gnocchi.xyz/stable_4.2/rest.html
- Gnocchi-Endpoint-URL (Servicetyp „Metric“) im Dashboard unter „API Access“ nachschlagen
- Oder Ermitteln der Endpoint-URL mittels der Dienstliste von OpenStack

```
List<? extends Service> catalog = client.identity().tokens().getServiceCatalog(client.getToken().getId())
```

→ Öffentlichen (Public) Endpoint des Servicetyps „Metric“ verwenden

- Authentifizierung bei Gnocchi-Anfragen erfolgt per HTTP-Header (Schlüssel-Wert-Paare)
 - Für *alle* Anfragen notwendig
 - Schlüssel (<key>): „X-Auth-Token“
 - Wert (<value>): Token von OpenStack anfordern

```
String authToken = client.getToken().getId();
```

- Header-Modifikation bei REST-Anfragen

```
Response r = target.request().header(<key>, <value>).post(Entity.text("test"));
```


- Instanz-spezifische ID einer Metrik (z. B. `cpu`) ermitteln (im Folgenden: `<metric-id>`)

→ GET-Anfrage auf Pfad listet alle Metriken auf:

`<Gnocchi-URL>/v1/resource/instance/<vm-id>`

- Rückgabe der Ergebnisse erfolgt im JSON-Format
- Datentyp: `MWGnocchiInstanceResource`

- Messwerte für eine bestimmte Metrik abfragen

→ GET-Anfrage auf Pfad:

`<Gnocchi-URL>/v1/metric/<metric-id>/measures?start=<time>&granularity=10&aggregation=mean`

- „`<time>`“: Zeitstempel (analog zu CloudWatch) **oder** relative Zeitangabe, z. B. „`-30seconds`“
- „`granularity=10`“: Jeweils über 10 Sekunden aggregierte Datenpunkte abrufen
 - OpenStack Ceilometer sammelt bei uns alle 10 Sekunden neue Daten
 - Mögliche Aggregationszeiträume: 10 / 60 / 3600 Sekunden
- „`aggregation=mean`“: Durchschnitt über Aggregationszeitraum
- Datentyp: `String[][]`; pro Array-Element: Zeitstempel, Aggregationszeitraum, Wert

- CPU-Metrik gibt akkumulierte Rechenzeit zurück

- „`aggregation=rate:mean`“: CPU-Verbrauch des aktuellen Aggregationszeitraums in Nanosekunden
- CPU-Auslastung: Messwert / Aggregationszeitraum
beispielsweise $7\,000\,000\,000\text{ ns} / 10\,000\,000\,000\text{ ns} = 70\%$