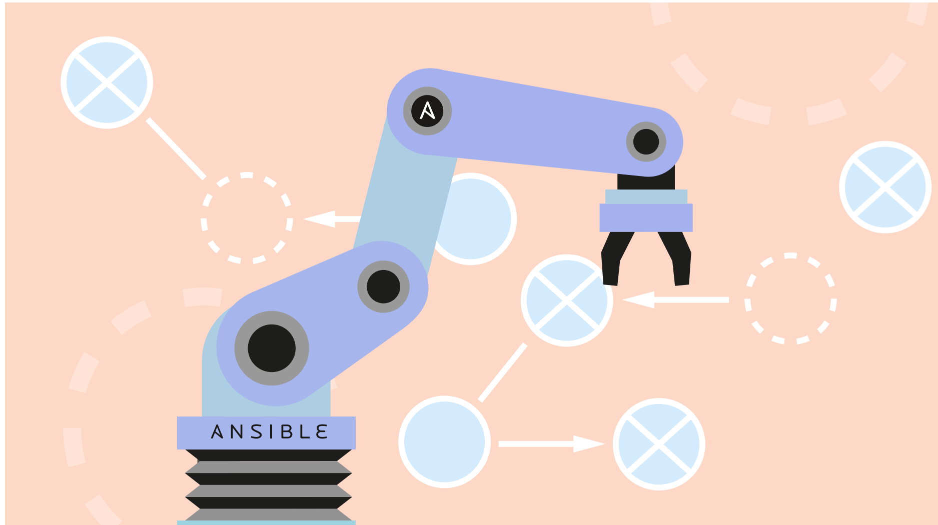


Event-Driven Ansible – automatisch automatisiert

Ansible hat sich als Werkzeug zur Konfiguration und Orchestrierung von IT-Systemlandschaften bewährt. Traditionell bestimmt der Administrator den Zeitpunkt der Ausführungen von Ansible Playbooks. Mit Event-Driven Ansible reagiert das Ansible-Ökosystem auf bestimmte Ereignisse automatisch.

Von Mark Pröhl und Philippe Schmid

Mit Ansible lassen sich praktisch alle Arten von Administrationsaufgaben in Linux- und Windows-Umgebungen automatisieren. Auch beim Verwalten von Netzwerkkomponenten und im Bereich der Security-Automation kommt die Software immer häufiger zum Einsatz. Doch haben bislang alle verfügbaren Werkzeuge gemeinsam, dass ein Benutzer den Zeitpunkt der Ausführung bestimmt. Sei es durch das Absetzen eines Kommandos im Terminal, durch Klicken des entsprechenden Knopfes in einer Software oder das Definieren eines Schedule (siehe Kasten „Ansible-Basics“). Bisher gab es im Ansible-Projekt keine mitgelieferten Möglichkeiten, die Ausführung eines Ansible Playbooks an das Zutreffen einer definierten Bedingung zu koppeln.

Diese Lücke schließt nun Event-Driven Ansible (EDA) mit seinem Kommandozeilenwerkzeug `ansible-rulebook` und der dazugehörenden Ansible Collection ansible.eda, die durch eine umfangreiche Projektdokumentation ergänzt werden (siehe ix.de/zez3). Das Projekt begann Anfang 2022 mit ersten Commits auf GitHub, auf dem Ansible Fest im Oktober 2022 wurde es offiziell vorgestellt.

Source-Plug-ins

Die neuen EDA-Werkzeuge erweitern Ansible um die Fähigkeit, Ansible Runs automatisiert auszuführen, wenn bestimmte Ereignisse eintreten und bestimmte Bedingungen erfüllt sind. Events haben ihren Ursprung in Sources, die in Form von Source-Plug-ins als Python-Programme implementiert sind.

Die Collection ansible.eda bietet bereits eine große Auswahl vordefinierter Source-Plug-ins für spezielle Anwendungsfälle. Zu reinen Debugging-Zwecken dient beispielsweise ansible.eda.range, ein Source-Plug-in, das numerische Events innerhalb eines ganzzahligen Bereichs produziert, oder ansible.eda.generic, mit dem sich beliebige Event-Nutzdaten einfach modellieren lassen. Beispiele für mitgelieferte Plug-ins mit praktischer Anwendbarkeit sind solche für Journald, Kafka oder den Prometheus Alertmanager. Ebenfalls enthält die Collection produktunabhängige Plug-ins wie das weiter unten vorgestellte ansible.eda.url_check, das die Verfügbarkeit einer Webseite prüft.

Es ist anzunehmen, dass der Hersteller und auch die Ansible-Community in Zukunft viele weitere Source-Plug-ins für speziellere Zwecke hervorbringen werden, so wie das bereits bei den Ansible-Modulen der Fall war. Auch hausinterne IT-Komponenten, für die die Collection ansible.eda kein passendes Source-Plugin bereitstellt, kann man als Eventquellen nutzen. Dazu nötige Custom-Plug-ins lassen sich, wie im Ansible-Umfeld üblich, auch bei EDA in Python implementieren.

Regelwerk

Die Steuerung von EDA erfolgt über ein in Rulebooks definiertes Regelwerk. Diese YAML-Dateien sind Ansible Playbooks sehr ähnlich. Rulebooks fassen ein oder mehrere sogenannte Rulesets zusammen, von denen wiederum jedes beschreibt, wie EDA auf Events aus den verschiedenen Quellen reagieren soll. In der YAML-Datei des Rulebooks definiert pro Ruleset das Schlüsselwort `sources:` die Eventquellen. Regeln, wie auf das Eintreten von Events reagiert werden soll, beschreibt der Abschnitt `rules:`.

Jede Regel besteht aus einer Bedingung (`condition:`) sowie einer Aktion

- Bei der Automatisierung mit Ansible ohne Zusatzsoftware müssen die in Playbooks beschriebenen Aufgaben immer noch benutzergesteuert angestoßen werden.
- Mit dem noch jungen Event-Driven Ansible (EDA) lässt sich das Ausführen von Automatisierungsabläufen an das Eintreten bestimmter Ereignisse koppeln.
- Hierbei beschreiben Rulebooks Regelwerke, um Ereignisse mit definierten Aktionen zu verknüpfen.
- EDA ermöglicht Operations as Code und Self-Healing-Szenarien und hebt damit die Systemautomatisierung auf ein neues Level.

Ansible-Basics

Die grundlegende Funktionsweise von Ansible lässt sich ist sehr schnell verstehen: Ausgangspunkt sind die Playbooks, in YAML geschriebene Dateien, die den Ablauf einer Automatisierung definieren. Auf zentralen Maschinen, den Control Nodes, analysiert Ansible solche Playbooks und generiert daraus Python-Skripte. Diese werden über einen Connection-Mechanismus (unter Linux ist das meistens SSH) auf das zu konfigurierende System kopiert, dort ausgeführt und anschließend wieder gelöscht. iX hat sich über den Jahreswechsel 2020/2021 in einem dreiteiligen Tutorial mit dem praktischen Ansible-Einsatz befasst [1], [2], [3].

Am einfachsten starten Ansible-Administratoren solche Runs, indem sie auf dem Control Node einen passenden Kommandozeilenbefehl (beispielsweise `ansible-playbook`) absetzen. Daneben ermöglichen speziell zu diesem Zweck geschriebene Programme elegantere Wege, um Ansible Runs auszuführen. Ein Beispiel ist das Open-Source-Projekt AWX, das auch die Basis für Red Hats Ansible Automation Platform (AAP) oder CIQs Ascender darstellt (siehe ix.de/zez3). Weitere Beispiele wären Ansible Semaphore oder das Einbinden von Ansible Runs in Tools wie Jenkins, GitHub und GitLab.

Mehr über Conditions

Der Abschnitt `condition:` innerhalb einer `rule:` regelt, wann die unter `actions:` definierten Aktionen ausgeführt werden sollen. Eine Condition kann eine oder mehrere Bedingungen enthalten, wobei eine beliebige (any), alle (`all`) oder nicht alle (`not_all`) zutreffen müssen. Eine Bedingung verknüpft mittels logischer Operatoren die Informationen aus einer der folgenden Quellen:

```
Events
Facts
Variablen
```

Auf das aktuelle Event greift man mit dem Präfix `event` zu. Beispielsweise nutzt die Condition aus Listing 5 `event.alert.labels.job` und `event.alert.status`, um Informationen des Prometheus Alertmanagers auszuwerten. Sind im Rulebook Informationen von mehreren Ereignissen verfügbar, kann man mit dem Präfix in der Mehrzahl auf die verschiedenen Ereignisse zugreifen, etwa `events.alert`. Setzt man im Rulebook den Parameter `gather_facts` auf `true`, sind über das Präfix `facts` die typischen Ansible Facts wie `ansible_distribution` zugänglich. So bewirkt beispielsweise `facts.ansible_distribution == RedHat` als Teil einer Bedingung, dass eine Aktion nur auf RedHat-Systemen ausgeführt wird. Über das Präfix `fact` kann eine Condition auf von anderen Regeln mit der Aktion `set_fact` definierte Inhalte zugreifen.

Mit dem Präfix `vars` wiederum lassen sich die mit der `ansible-rulebook`-Kommandozeilenoption `--vars` übergebenen Variablen verwenden. Definiert man beispielsweise eine Variable `foo` in einer YAML-Datei und lädt diese per `--vars` zur Laufzeit, so referenziert `vars.foo` deren Wert, wie das folgende Beispiel zeigt:

Variablen-Beispiel

```
name: nur eine der Bedingungen ↵
                    muss zutreffen
condition:
  any:
    - event.url_check.status ↵
                           == "down"
    - facts.ansible_distribution ↵
                is regex("redhat",↵
                   ignorecase=true)
    - vars.foo is not defined
```

(`action:`). Nicht nur Events können Bedingungen sein, sondern auch zuvor gespeicherte Facts und Variablen. Eine Condition kann wiederum mehrere andere enthalten, wobei eine, mehrere oder alle zutreffen müssen. In einer Bedingung lassen sich die gängigen Datentypen wie Integer, String und Boolean über logische Operatoren miteinander verknüpfen (siehe auch Kasten „Mehr über Conditions“).

`action` wiederum definiert, was genau zu tun ist: Mit `run_module` kann man Ad-hoc-Kommandos ausführen – also einzelne Ansible-Module. `run_playbook` spielt ganze Playbooks ab. Und schließlich kann `run_job_template` Ansible-Job-Templates in einer AWX-, AAP- oder Ascender-Instanz anstoßen. Neben diesen Actions gibt es noch solche wie `set_fact` oder `shutdown`, die weitere Möglichkeiten zur Steuerung der Ablauflogik eines Rulebooks eröffnen.

Intern verwendet `ansible-rulebook` das in Java geschriebene Decision-Making-Framework drools. Dies bedeutet, dass neben der Installation von `ansible-rulebook` und der Collection ansible.eda auch eine Java-Runtime-Umgebung vorhanden sein muss. Der Kasten „EDA-Installation im Detail“ beschreibt den Vorgang genauer.

EDA-Installation im Detail

Um Event-Driven Ansible verwenden zu können, muss man das Python-Paket ansible-rulebook zusammen mit der Ansible Collection ansible.eda installieren. Da ansible-rulebook das Java-Framework drools verwendet, muss außerdem OpenJDK installiert sein. Darüber hinaus benötigt ansible-rulebook das Python-Paket jpy. Auf RHEL-kompatiblen Distributionen empfiehlt sich die Installation der abhängigen Pakete über den Paketmanager `dnf`, wohingegen man ansible-rulebook selbst besser per `pip` installiert. Das Listing „EDA-Installationsschritte“ zeigt das Vorgehen, indem es zusätzlich eine virtuelle Python-Umgebung (`venv`) unter ~/python/bin/activate einrichtet. Die drei Pakete systemd-devel, gcc und python3-devel werden schließlich für den letzten Installationsschritt benötigt.

EDA-Installationsschritte

```
sudo dnf install java-17-openjdk python3-pip
python3 -m venv ~/python
. ~/python/bin/activate
pip install --upgrade pip
pip install ansible ansible-rulebook
ansible-galaxy collection install ansible.eda
sudo dnf install systemd-devel gcc python3-devel
pip install -r ~/.ansible/collections/ansible_collections/ansible/eda/↵
                                                              requirements.txt
```

Listing 1: Das Rulebook webserver_rulebook.yml

```
- name: Neustart Webserver wenn Seite down
  hosts: node1.example.com
  sources:
    - ansible.eda.url_check:
        urls:
          - http://node1.example.com:80/
  rules:
    - name: Test ob Webseite verfügbar und Neustart, wenn nicht
      condition: event.url_check.status == "down"
      action:
        run_playbook:
          name: start_webserver.yml
```

Listing 2: Das simple Playbook start_webserver.yml

```
- hosts: node1.example.com
  become: true
  tasks:
    - name: start and enable httpd
      ansible.builtin.service:
        name: httpd
        state: started
```

Listing 3: Das Beispiel-Inventory company_inventory

```
[controller]
control.example.com

[web]
node1.example.com
node2.example.com
```

Der Programmcode von `ansible-rulebook` läuft typischerweise auf einem Control Node, also einer Maschine, die auch Ansible Playbooks ausführt. Denn startet ein Rulebook eine Aktion per `run_playbook` oder `run_module`, muss der `ansible-rulebook`-Prozess auf demselben Server laufen wie das auszuführende Playbook oder Modul. Ausnahme: Beim Einsatz der Aktion `run_job_template` kann `ansible-rulebook` auf einem anderen Server als dem Control Node laufen. Die Ausführung des Job-Templates wird dann über die API des AWX-, AAP- oder Ascender-Controller gestartet. Diese Architektur ist in der neusten Version von AAP mit dem Ansible-Controller und dem EDA-Controller bereits abgebildet.

Im Gegensatz zu `ansible-playbook` verrichtet `ansible-rulebook` seine Arbeit als Daemon: Einmal gestartet, läuft der Prozess so lange im Hintergrund, bis eines der Source-Plug-ins endet. Sonst wird er nur im Fehlerfall oder bei händischem Einwirken beendet.

Ein einfaches Beispiel

Als erstes Beispiel für den Einsatz von EDA soll hier die Überwachung einer Website dienen. Das Rulebook webserver_rulebook.yml (Listing 1) prüft die Verfügbarkeit einer Webseite auf dem Server node1.example.com und führt im Fehlerfall das Ansible Playbook start_webserver.yml (Listing 2) aus. Dieses Playbook ist bewusst einfach gehalten und soll lediglich sicherstellen, dass ein Webserver läuft. Eine Übersicht über die Beispielumgebung zeigt die Inventory-Datei in Listing 3.

Das Rulebook nutzt zur Verfügbarkeitsprüfung das Source-Plug-in ansible.eda.url_check. Wie der Name erkennen lässt, stammt dieses aus der Ansible Collection ansible.eda. Die einzige Regel unterhalb von `rules:` definiert, dass die Action `run_playbook` das Playbook start_webserver.yml starten soll, sollte der Test im Source-Plug-in den Status `down` melden.

Im Beispiel läuft das ausgeführte Playbook auf allen Servern der Inventary-Gruppe `web` los, nicht nur auf Servern mit der zu überwachenden Website. Um das zu optimieren, nutzt man das Keyword `hosts:` in der Regeldefinition, um einzugrenzen, wo die `action` ausgeführt wird. Das heißt, dass die im Playbook start_webserver.yml definierten Hosts weiter auf die im Rulebook definierten eingegrenzt werden.

Ausführen lässt sich das Rulebook mit dem Kommando `ansible-rulebook` unter Angabe des zu verwendenden Inventorys `-i`:

```
ansible-rulebook --rulebook ↵
        webserver_rulebook.yml -i ↵
        company_inventory --verbose
```

Die zusätzliche Option `--verbose` ist für Testläufe sehr hilfreich: Sie ermöglicht es, im Output des Kommandos mitzuverfolgen, welchen Status der `ansible-rulebook`-Daemon zu jedem Zeitpunkt hat. So kann man die Reaktion auf einen Ausfall der Webseite im Beispiel in Echtzeit beobachten. Tritt nun eine `condition` ein, so wird das Playbook ausgeführt und der `ansible-rulebook`-Prozess läuft im Hintergrund weiter.

Das lässt sich einfach testen: Stoppt man per `systemctl stop httpd.service` den Webserver auf node1, so kann man im Output des mit der Option `--verbose` laufenden `ansible-rulebook`-Prozesses mitverfolgen, wie EDA die Situation erkennt und als Gegenmaßnahme das Playbook start_webserver.yml startet (Listing 4). Ein anschließender Test auf node1 mit `systemctl status httpd.service` sollte den Erfolg bestätigen.

EDA prüft den Zustand der `source` standardmäßig alle zehn Sekunden. Im

Listing 4: Auszug der Konsolenmeldungen

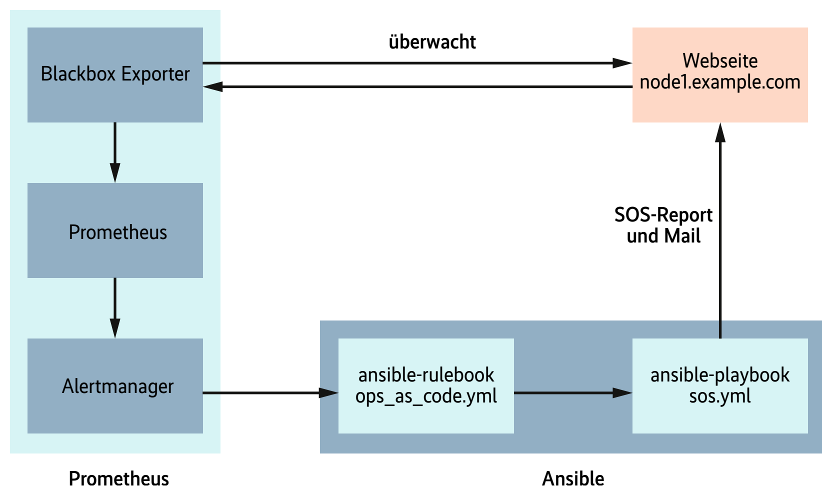
```
2023-07-19 07:06:27 518 [main] INFO org.drools.ansible.rulebook.integration.api.rulesengine.RegisterOnlyAgendaFilter - Activation of ↵
   effective rule "Test ob Webseite verfügbar und Neustart wenn nicht" with facts: {m={url_check={error_msg=Cannot connect to host node1.↵
               example.com:80 ssl:default [Connect call failed ('192.168.200.105', 80)], url=http://node1.example.com:80/, status=down}, ↵
                         meta={received_at=2023-07-19T11:06:27.518081Z, source={name=ansible.eda.url_check, type=ansible.eda.url_check}, ↵
                                                                                          uuid=b74ec6b7-bfb6-4aa5-b22d-f773ea0c6e8f}}}
2023-07-19 07:06:27,519 - ansible_rulebook.rule_generator - INFO - calling Test ob Webseite verfügbar und Neustart wenn nicht
2023-07-19 07:06:27,519 - ansible_rulebook.rule_set_runner - INFO - call_action run_playbook
2023-07-19 07:06:27,519 - ansible_rulebook.rule_set_runner - INFO - substitute_variables [{'name': 'start_webserver.yml'}] [{'event': ↵
    {'url_check': {'error_msg': "Cannot connect to host node1.example.com:80 ssl:default [Connect call failed ('192.168.200.105', 80)]", ↵
     'url': 'http://node1.example.com:80/', 'status': 'down'}, 'meta': {'received_at': '2023-07-19T11:06:27.518081Z', 'source': {'name': ↵
                             'ansible.eda.url_check', 'type': 'ansible.eda.url_check'}, 'uuid': 'b74ec6b7-bfb6-4aa5-b22d-f773ea0c6e8f'}}}]
2023-07-19 07:06:27,519 - ansible_rulebook.rule_set_runner - INFO - action args: {'name': 'start_webserver.yml'}
2023-07-19 07:06:27,519 - ansible_rulebook.builtin - INFO - running Ansible playbook: start_webserver.yml
2023-07-19 07:06:27,520 - ansible_rulebook.builtin - INFO - ruleset: Neustart Webserver wenn Seite down, rule: Test ob Webseite ↵
                                                                                                        verfügbar und Neustart wenn nicht
2023-07-19 07:06:27,520 - ansible_rulebook.builtin - INFO - Calling Ansible runner

PLAY [web] *********************************************************************

TASK [Gathering Facts] *********************************************************
ok: [node2.example.com]
ok: [node1.example.com]

TASK [start and enable httpd] **************************************************
2023-07-19 07:06:28 519 [main] INFO org.drools.ansible.rulebook.integration.api.rulesengine.RegisterOnlyAgendaFilter - Activation of ↵
   effective rule "Test ob Webseite verfügbar und Neustart wenn nicht" with facts: {m={url_check={error_msg=Cannot connect to host node1.↵
               example.com:80 ssl:default [Connect call failed ('192.168.200.105', 80)], url=http://node1.example.com:80/, status=down}, ↵
                         meta={received_at=2023-07-19T11:06:28.519433Z, source={name=ansible.eda.url_check, type=ansible.eda.url_check}, ↵
                                                                                          uuid=519fa55d-8bc1-453b-9636-df52b2eda9ef}}}
2023-07-19 07:06:28,520 - ansible_rulebook.rule_generator - INFO - calling Test ob Webseite verfügbar und Neustart, wenn nicht
ok: [node2.example.com]
changed: [node1.example.com]

PLAY RECAP *********************************************************************
node1.example.com          : ok=2    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
node2.example.com          : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
2023-07-19 07:06:28,833 - ansible_rulebook.builtin - INFO - Ansible Runner Queue task cancelled
2023-07-19 07:06:28,833 - ansible_rulebook.builtin - INFO - Playbook rc: 0, status: successful
2023-07-19 07:06:28,833 - ansible_rulebook.rule_set_runner - INFO - Task action::run_playbook::Neustart Webserver wenn Seite down::Test ↵
                                                                  ob Webseite verfügbar und Neustart wenn nicht finished, active actions 0
2023-07-19 07:06:28,834 - ansible_rulebook.rule_set_runner - INFO - call_action run_playbook
2023-07-19 07:06:28,834 - ansible_rulebook.rule_set_runner - INFO - substitute_variables [{'name': 'start_webserver.yml'}] [{'event': ↵
    {'url_check': {'error_msg': "Cannot connect to host node1.example.com:80 ssl:default [Connect call failed ('192.168.200.105', 80)]", ↵
     'url': 'http://node1.example.com:80/', 'status': 'down'}, 'meta': {'received_at': '2023-07-19T11:06:28.519433Z', 'source': {'name': ↵
                             'ansible.eda.url_check', 'type': 'ansible.eda.url_check'}, 'uuid': '519fa55d-8bc1-453b-9636-df52b2eda9ef'}}}]
2023-07-19 07:06:28,834 - ansible_rulebook.rule_set_runner - INFO - action args: {'name': 'start_webserver.yml'}
2023-07-19 07:06:28,834 - ansible_rulebook.builtin - INFO - running Ansible playbook: start_webserver.yml
2023-07-19 07:06:28,835 - ansible_rulebook.builtin - INFO - ruleset: Neustart Webserver wenn Seite down, rule: Test ob Webseite ↵
                                                                                                       verfügbar und Neustart, wenn nicht
2023-07-19 07:06:28,835 - ansible_rulebook.builtin - INFO - Calling Ansible runner
```

Rulebook lässt sich dieses Zeitintervall mit dem Schlüsselwort `delay` überschreiben. Dabei gilt es aber auch zu bedenken, wie lange die vollständige Ausführung der später verwendeten Action dauert. Allenfalls ist die Häufigkeit, mit der die `condition` evaluiert wird, mit der Direktive `throttle` zu drosseln. So lässt sich sicherstellen, dass keine Action mehrmals gleichzeitig läuft.

Operations as Code

EDA ist noch relativ jung. Die Zukunft wird zeigen, wo seine Stärken liegen und in welcher Art und Weise es sich am besten einsetzen lässt. Beispielsweise könnte EDA seine Stärken dann ausspielen, wenn es zusätzliche Instanzen einer Ressource zur Verfügung stellen soll. Auch für die Steuerung von Failover-Szenarien scheint EDA prädestiniert: Beim Ausfall eines Systems könnte es automatisch Ersatzressourcen starten. Allerdings bleiben derartige automatische Fehlerbehebungen (Self-Healings) grundsätzlich ein schwieriges Unterfangen, da sich im Vorfeld nie alle möglichen Fehlerquellen erfassen lassen.

Die Zukunft von Event-Driven Ansible könnte daher eher darin liegen, in ei-



Im Artikelbeispiel für Operations as Code überwacht Prometheus eine Webseite und stößt bei deren Ausfall über EDA eine Alarmierung und das Erstellen eines SOS-Reports für den Support an.

Prometheus und Alertmanager als Eventquelle einrichten

Prometheus ist ein Open-Source-Monitoring- und -Alerting-Werkzeug. In erster Linie sammelt es Metrikinformationen. Mit einem „Blackbox Exporter“ kann es auch Verfügbarkeiten von Webseiten überwachen. Der Alertmanager ist die Prometheus-Komponente, die zur Benachrichtigung von externen Stellen wie Event-Driven Ansible zuständig ist.

Das Source-Plug-in ansible.eda.alertmanager erstellt auf dem EDA-Server einen Webhook, der auf dem definierten Port (hier 5000) nach eingehenden Nachrichten horcht. Die Firewall darf also die Verbindungsanfragen nicht blockieren. Außerdem wird die Meldung vom Alertmanager unter https://<eda-controller>:5000/endpoint erwartet. Entsprechend ist die URL im Alertmanager zu konfigurieren. Eine Beispielkonfiguration /etc/prometheus/alertmanager.yml des Alertmanagers könnte wie im Listing „Prometheus-Beispiel“ aussehen.

Prometheus-Beispiel

```yaml
global:
  resolve_timeout: 3m

receivers:
- name: eda
  webhook_configs:
  - url: http://<eda-controller>:↵
                    5000/endpoint

inhibit_rules:
  - source_match:
      severity: 'critical'
    target_match:
      severity: 'warning'
    equal:
      - alertname
      - dev
      - instance

route:
  group_by:
  - alertname
  group_wait: 10s
  group_interval: 10s
  receiver: 'eda'
  repeat_interval: 1h
```

nem Fehlerfall gewisse Administrationsaufgaben automatisch durchzuführen. Das dazugehörende englische Schlagwort lautet „Operations as Code“. Der folgende Abschnitt soll die Möglichkeiten von EDA in diesem Bereich anhand eines Praxisbeispiels aus dem Alltag des Linux-Betriebs aufzeigen.

Bei Problemen mit einem auf einem RHEL-System betriebenen Service besteht einer der ersten Schritte oft im Anfertigen eines SOS-Reports. Dieser enthält sämtliche relevanten Logdateien und sollte im Rahmen von Supportanfragen beim Hersteller griffbereit sein. Um Zeit und manuelle Arbeit zu sparen, drängt es sich auf, diesen Schritt per EDA zu automatisieren.

Im Beispiel übernimmt das weitverbreitete Monitoringsystem Prometheus die Aufgabe der Überwachung und sein Alertmanager soll in EDA als Eventquelle eingebunden werden (siehe Kasten: „Prometheus und Alertmanager als Eventquelle einrichten“). Das Rulebook ops_as_code.yml aus Listing 5 enthält alles, um diesen Schritt zu automatisieren. Trifft eine Warnmeldung des Überwachungssystems ein, soll EDA auf dem fehlerhaften Server einen SOS-Report ausführen und diesen auch gleich automatisch an ein (fiktives) Ticketsystem weiterleiten.

Das Rulebook in Listing 5 verwendet das Source-Plug-in `alertmanager`, das Alarme von Prometheus entgegennehmen kann. Als Action wird das in Listing 6 dargestellte Playbook sos.yml ausgeführt, falls die vom Alertmanager gesendete Meldung gewissen Bedingungen genügt: Die Meldung muss den Status `firing` und das Job-Label `blackbox` beinhalten. Status und Job-Label sind Informationen, die Prometheus liefert. Jedes Source-Plug-in liefert andere Informationen, um die herum sich die Conditions bauen lassen.

Die eigentliche Aufgabe, den SOS-Report zu erstellen, übernimmt das Playbook sos.yml aus Listing 6. Dieses stellt zunächst sicher, dass auf dem fehlerhaften Server das DNF-Paket sos installiert ist, und führt dann das Kommando `sos` über das Modul ansible.builtin.command aus. Dabei soll dieses Kommando unattended, also ohne interaktive Eingabe von Parametern erfolgen, wozu die Option `--batch` dient.

Weiterhin soll der SOS-Report ein Label erhalten, damit es sich später dem auslösenden Prometheus-Alarm einfach zuordnen lässt. Die Option `--label` gibt der Warnung dazu den Wert von `alertname` mit. EDA stellt dem ausgeführten Playbook zu diesem Zweck Informationen zu Events, Facts und anderen Variablen zur Verfügung. Diese EDA-Informationen

Listing 5: Das Rulebook ops_as_code.yml

```yaml
- name: Prometheus Alertmanager
  hosts: node1.example.com
  sources:
    - ansible.eda.alertmanager:
        host: 0.0.0.0
        port: 5000
  rules:
    - name: Erstelle SOS-Report wenn Alert gemeldet wird
      condition:
        all:
          - event.alert.labels.job == "blackbox"
          - event.alert.status == "firing"
      throttle:
        once_within: 5 minutes
        group_by_attributes:
          - event.meta.source.type
      actions:
        - run_playbook:
            name: sos.yml
```

Listing 6: Auszug aus sos.yml für den SOS-Report

```yaml
- hosts: node1.example.com
  become: true
  tasks:
    - name: Installiere sos Paket
      ansible.builtin.dnf:
        name:
          - sos
        state: installed

    - name: Entferne bestehende SOS Reports
      ansible.builtin.shell:
        cmd: "rm -f /var/tmp/sosreport*"

    - name: Erstelle einen SOS Report mit Label alertname
      ansible.builtin.command:
        argv:
          - "/usr/sbin/sos"
          - "report"
          - "--clean"
          - "--batch"
          - "--label"
          - "{{ ansible_eda.event.payload.alerts[0].labels.alertname  }}"

    - name: Sende den SOS-Report an das TicketSystem
      community.general.mail:
        #...
        #/gekürzt/
        #...
```

iX-Workshop zu IT-Automatisierung mit Event-Driven Ansible

Der zweitägige Workshop geht auf eine Reihe von Fragen und Praxisproblemen ein, die erfahrungsgemäß bei der Einführung der Ansible Automation Platform (AAP) von Red Hat auftreten. Dabei erhalten die Teilnehmenden eine fundierte Einführung in das relativ neue Thema Event-Driven Ansible (EDA).

Sie lernen, wie Ansible mithilfe von Rulebooks auf bestimmte Ereignisse in der IT-Umgebung durch vordefinierte Aktionen automatisiert reagiert und so den Betrieb der IT-Infrastruktur auch im Störungsfall aufrechterhalten kann.

Trainer: Mark Pröhl, Daniel Kobras

Der Workshop findet online statt. Termine und weitere Informationen gibt es unter heise.de/s/k0vK.

kann man im Playbook als Teil der Ansible-Variablenstruktur mit dem Präfix `ansible_eda` ansprechen. Der Name des Alarms findet sich so in `ansible_eda.event.payload.alerts[0].labels.alertname` und kann dem `sos`-Kommando übergeben werden.

Der generierte SOS-Report wird dann über das Modul community.general.mail direkt an ein fiktives Ticketing-Tool geschickt. Das Erstellen des SOS-Reports und das Versenden der E-Mail kann einige Minuten dauern. Hier besteht also Potenzial für Race Conditions. Zum Vermeiden, dass EDA das Playbook sos.yml mehrfach gleichzeitig startet, dient die Anweisung `throttle:` in Listing 5. Diese drosselt die Evaluierung der Bedingung derart, dass höchstens alle fünf Minuten ein neuer Playbook-Run ausgeführt wird. Auch hier könnten weitere Optimierungen angebracht sein.

Die EDA-Dokumentation erläutert `throttle` mit den Optionen `once_within` und `group_by_attributes` im Detail (siehe ix.de/zez3). Der Kasten „Mehr über Conditions“ gibt einen Überblick über die Möglichkeiten, wie sich Conditions über logische Operationen verknüpfen lassen.

Ausblick

Event-Driven Ansible ist eine sinnvolle Erweiterung für Ansible. Viele Unternehmen starten Ansible Runs schon jetzt über GitLab-Pipelines und GitHub Actions. Deren Logik ähnelt der von EDA: Ein Ansible Run wird ausgelöst, wenn gewisse Ereignisse wie Pull Requests oder Merge Operations eintreten. Allerdings sind die EDA -Regelwerke wesentlich flexibler.

Welche Hersteller Source-Plug-ins für ihre Produkte anbieten, wird die Einsatzmöglichkeiten von EDA stark beeinflussen. Red Hat hat die Zusammenarbeit mit vielen namhaften Unternehmen für die Einbindung seiner Produkte über Source-Plug-ins zugesagt (siehe ix.de/zez3). Bei den Anbietern von Netzwerkgeräten sind es unter anderen Palo Alto Networks, F5 und Cisco. Source-Plug-ins für Monitoringlösungen gibt es neben dem Prometheus Alertmanager auch für Zabbix und Sensu. Viele weitere Source-Plug-ins, beispielsweise für AppDynamics, Dynatrace und ServiceNow, werden laut Red Hat bald erhältlich sein.

In den Enterprise-Produkten zu Ansible ist EDA bereits angekommen: Die neusten Versionen von Red Hats Ansible Automation Platform, CIQs Ascender und auch AWX haben EDA schon integriert. Die Einführung von Decision Environments zur Ausführung von `ansible-rulebook` wie auch die Integration in eine übersichtliche grafische Oberfläche sind sicherlich wichtige Voraussetzungen dafür, dass sich EDA auch im Enterprise-Umfeld durchsetzen wird. (avr@ix.de)

Quellen

[1] Daniel Kobras, Mark Pröhl; Ansible-Tutorial, Teil 1: Custom Inventories, iX 11/2020, S. 123

[2] Daniel Kobras, Mark Pröhl; Ansible-Tutorial, Teil 2: Variables, Facts und Debugging; iX 12/2020, S. 126

[3] Daniel Kobras, Mark Pröhl; Ansible-Tutorial, Teil 3: Module und Collections; iX 1/2021, S. 130

[4] Links zu Hintergrundinformationen und zur EDA-Onlinedokumentation: ix.de/zez3

MARK PRÖHL

ist Principal Architect bei Puzzle ITC Deutschland und beschäftigt sich dort unter anderem mit Digital Identity sowie DevOps-Themen und Automatisierung.

PHILIPPE SCHMID

ist Co-Leader des Teams System Engineering bei Puzzle ITC Schweiz und beschäftigt sich dort vorwiegend mit Ansible-Automatisierung.