12 Icinga-DSL

Die DSL von Icinga 2 ist eine mächtige Konfigurationssprache und damit leider auch anfällig gegenüber Fehlern, die einem beim Schreiben unterlaufen. In diesem Kapitel wird auf erweiterte Möglichkeiten für Tests und Debugging eingegangen – die Console.

Ebenfalls wird gezeigt, wie eigene Funktionen geschrieben und getestet werden können. Eine Erweiterung der DSL bieten Schleifen, die es ermöglichen, Iterationen abzubilden. Wichtig im Zusammenhang mit Funktionen, Objekten und Schleifen sind die Gültigkeitsräume von Variablen, auf die in diesem Kapitel ebenfalls eingegangen wird.

»Das Leben bewegt sich sehr, sehr schnell.
Wenn du nicht gelegentlich anhältst und dich umschaust, könntest
du es verpassen.«

Ferris Bueller, 1986

12.1 Console

Der CLI-Befehl *console* kann zum Debuggen der Icinga-Konfiguration eingesetzt werden. Er lässt sich aber auch zum Testen und Auswerten von z. B. Funktionen in einer lokalen Sandbox verwenden.

Möchte man etwas testen, was nichts mit einer laufenden Konfiguration zu tun hat, reicht ein einfacher Aufruf:

```
$ icinga2 console
Icinga 2 (version: 2.13.2-1)
Type $help to view available commands.
<1> =>
```

Die Console meldet sich mit der aktuellen Version von Icinga 2, einem Hinweis, wie zum Einstieg Hilfestellungen in Anspruch genommen werden können, und abschließend mit einem Prompt.

```
<1> => h=4m
null
<2> => h
240.000000
```

Eine Variablenzuweisung kann selbstverständlich mit allen bekannten Datentypen geschehen. Die erfolgreiche Zuweisung wird mit der Ausgabe von *null* quittiert, andernfalls wird eine Fehlermeldung angezeigt. Die Ausgabe des Inhalts einer Variablen erfolgt durch Aufruf der Variablen. Das obige Beispiel zeigt, dass Icinga intern bei Zeitintervallen immer mit Sekunden arbeitet.

Icinga kennt Referenzen. So wird eine Änderung des Inhalts an das referenzierte Objekt weitergegeben bzw. natürlich dessen Speicherinhalt angepasst:

```
<3> => r=&h
null
<4> => *r=2h
null
<5> => h
7200.000000
```

Zum Verlassen der Console reicht ein altbekanntes CTRL-C, ein Kommando wie *exit* oder *quit* existiert nicht.

Über die API kann sich die Console auch mit einer laufenden Icinga-Instanz verbinden und damit auf alle deren Informationen zugreifen. Hierfür ist ein API-User erforderlich. Da auch hier keine Manipulationen an der laufenden Konfiguration vorgenommen werden können, kann ohne Weiteres auf einen Benutzer zurückgegriffen werden, der weitreichende Berechtigungen hat. In diesem Kapitel wird deswegen mit dem Api-User *root* gearbeitet.

```
object ApiUser "root" {
  password = "XXX"
  permissions = [ "*", ]
}
```

Codebeispiel 12.1: */etc/icinga2/zones.d/api-users.conf*

Nach einem Reload von Icinga 2 erfolgt mit einem Connection-String *https://user:password@host:port/* eine Verbindung mit der entsprechenden Instanz über die API. Alternativ lassen sich Benutzername und Passwort als Umgebungsvariable in der aktuellen Shell setzen. Beide werden dann automatisch zur Anmeldung verwendet und es reicht, Host und Port im Connection-String anzugeben.

```
$ export ICINGA2_API_USER=root
$ export ICINGA2_API_PASSWORD=XXX
$ icinga2 console --connect 'https://fornax.gwc-jonas.local:5665/'
```

Ist man korrekt authentifiziert, kann nun auf alle Variablen, Konstanten und Objekte und deren Eigenschaften zugegriffen werden.

```
<1> => NodeName
"fornax.gwc-jonas.local"
<2> => get_host(NodeName).zone
"main"
<3> => if (get_host(NodeName).zone == ZoneName) {"belongs to zone"}
"belongs to zone"
```

Aber nicht nur auf die schon aus den Konfigurationsdateien bekannten Eigenschaften eines Objekts wie einen Host kann zugegriffen werden, sondern insbesondere auch auf seine sich zur Laufzeit ändernden Eigenschaften, wie z. B. den Zeitpunkt des letzten Checks oder dessen ermittelten Status.

```
<4> => DateTime(get_host(NodeName).last_check).to_string()
"2021-11-14 14:37:42 +0100"
<5> => get_service(NodeName, "ping4").state
0.000000
```

In diesem Zusammenhang sei hier nochmals auf die Onlinedokumentation[1] verwiesen. Unterhalb von »Library Reference« sind alle diese nützlichen Funktionen ausführlich beschrieben, so auch die Funktion *DateTime*, die einen Unix-Zeitstempel in ein für Menschen lesbares Format umwandelt.

[1] https://icinga.com/docs/icinga-2

Selbstverständlich lassen sich solche Ergebnisse in Variablen speichern, um zu einem späteren Zeitpunkt wieder darauf zuzugreifen:

```
<6> => h=get_service(NodeName, "ping4")
<7> => h.last_check_result.performance_data
[ "rta=0.084000ms;100.000000;200.000000;0.000000", "pl=0%;5;15;0" ]
<8> => h.last_check_result.state
"0.000000"
<9> => h.last_check_result.state=2
null
<10> => h.last_check_result.state
"2.000000"
```

Obwohl die Dokumentation besagt, bei der Rückgabe der Zugriffsfunktionen auf Objekte handele es sich um eine Referenz können auch mit der Notation für Variablen und Referenzen keine Attribute des Objekts in der aktuell laufenden Konfiguration geändert werden. Es wird demnach auch weiterhin in einer Sandbox gearbeitet.

12.2 Schleifen und Iterationen

Schleifen über Objekte mittels *apply* im Zusammenspiel mit *for* sind uns schon bekannt. Die gleiche Notation wird auch außerhalb von Objektdefinitionen verwendet.

for loops

Die »for«-Schleife wird verwendet, um durch die Elemente eines Arrays zu iterieren. Die Icinga-eigene Funktion *log* schreibt den übergebenen String als Eintrag in die Logdatei von Icinga 2.

```
var list = [ "a", "b", "c" ]

for (item in list) {
  log("Item: " + item)
}
```

Der Block wird für jedes Element im Array einmal ausgewertet. Die Variable *item* wird als lokale Variable für die Schleifendurchläufe deklariert.

Ein Dictionary kann ebenfalls mit der bekannten Notation einer »foreach«-Schleife iterativ durchlaufen werden:

```
var dict = { a = 3, b = 7 }

for (key => value in dict) {
  log("Key: " + key + ", Value: " + value)
}
```

Mit *continue* und *break* lässt sich steuern, wie die Schleife ausgeführt wird. Mit *continue* werden alle restlichen Ausdrücke und Anweisungen im Block der Schleife übersprungen und es beginnt die nächste Auswertung, *break* hingegen bricht die Schleife komplett ab und es erfolgen keine weiteren Durchläufe.

while loop

Der »while«-Schleife wird ein boolescher Ausdruck mitgegeben. Die Schleife wird so lange durchlaufen, wie dieser Ausdruck wahr ist – das folgende Beispiel also genau fünfmal.

```
var num = 5

while (num > 0) {
  log("Number: " + num)
  num -= 1
}
```

Die beiden Anweisungen *continue* und *break* erfüllen dieselben Zwecke wie bei den »for«-Schleifen: *continue* überspringt die dahinter stehenden Anweisungen im Block und der nächste Durchlauf beginnt unmittelbar, *break* verlässt die Schleife und es erfolgen keine weiteren Iterationen.

12.3 Funktionen

Für das Verständnis und den Einsatz von Funktionen hilft die Kenntnis von anderen Programmier- und Skriptsprachen, die vergleichbare Konstrukte verwenden. Da Funktionen teilweise sehr komplex sind und oft auch nur für Spezialfälle genutzt werden, soll hier nicht allzu sehr in die Tiefe gegangen werden. Im Gegensatz zu den bisher beschriebenen Möglichkeiten können Funktionen nicht anhand einiger Beispiele erlernt werden, sondern erfordern meist ein tieferes Verständnis, als sich im Rahmen eines Buchs über ein Monitoring-System vermitteln lässt.

12.3.1 Definition eigener Funktionen

Eigene Funktionen werden mit *function* definiert, gefolgt von einem eindeutig zu vergebenen Namen und optional einer Parameterliste. So definiert

```
function percent(a, b) {
  return b * a / 100
}
```

die Funktion *percent*, die den prozentualen Anteil von *a* an *b* zurückliefert. Enthält eine Funktion kein explizites *return*, wird das Resultat des letzten Ausdrucks zurückgeliefert.

Ist obige Funktion in der laufenden Konfiguration definiert, lässt sie sich in der Console auf Korrektheit testen:

```
<1> => percent(10, 200)
20.000000
```

Um eine Funktion auch sicher aus einem Objekt heraus aufrufen zu können, sollte sie unbedingt als globale Funktion definiert werden. Hierbei muss eine alternative Syntax verwendet werden, die selbstverständlich auch für nicht globale Funktionen angewandt werden darf.

```
globals.percent = function(a, b) {
  b * a / 100
}
```

Codebeispiel 12.2: */etc/icinga2/zones.d/global-templates/functions.conf*

Das dem Namen vorangestellte »Schlüsselwort« *globals* spezifiziert den Gültigkeitsraum dieser Funktion als global. Ab nun ist gesichert, dass *percent* aus Objekten aufgerufen werden kann und sich Attributen oder Custom Variables zuweisen lässt. Ist der Rückgabewert ein Boolean kann die Funktion auch mittels *set_if* eingesetzt werden.

Ein Beispiel für den Einsatz der Funktion *percent* wäre das Setzen einer Obergenze (*vars.max_procs*) von gleichzeitig laufenden Prozessen am Host, z. B. in einem Template. An den Hosts oder in der Servicedefinition sähe der Aufruf dann so aus:

```
vars.procs_warning = percent(80, host.vars.max_procs)
vars.procs_critical = percent(90, host.vars.max_procs)
```

Damit gilt nach einem Reload, dass von jetzt an die Schwellenwerte prozentual zur angegebenen Obergrenze berechnet und gesetzt werden.

12.3.2 Lambda-Funktionen

Besonders kurze Definitionen von Funktionen können in der sogenannten Lambda-Syntax vorgenommen werden. Da sie sehr kurz gehalten sind, sind sie oft sehr schwer nachzuvollziehen. Als generelle Empfehlung zu Funktionen in der Lambda-Form kann gelten: Wer solche Konstrukte aus anderen Sprachen kennt und auf Anhieb anzuwenden weiß, kann sie verwenden. Icinga ist aber der falsche Ort, um deren Anwendung zu lernen. Die bisher besprochene Definition und die folgende verkürzte Lambda-Schreibweise ermöglichen alles, was die Lambda-Form bietet, nur in anderer Schreibweise, die leichter verständlich ist.

Die Notation der Lambda-Form legt erst Variablen für die zu übergebenen Werte fest und dann die eigentliche Funktion. Ein Beispiel zur Berechnung einer Quadratzahl in der Console sieht dabei wie folgt aus:

```
<1> => f=(x)=>x*x
null
<2> => f(2)
4.000000
```

Dabei wird der Funktion *f* ein Wert übergeben und das errechnete Quadrat dieser Zahl zurückgegeben. Anstelle von *f* und *x* können natürlich auch sprechende Namen wie *sqr* als Namen gewählt werden.

Genau wie bei »normalen« Funktionen müssen auch Lambda-Funktionen global definiert werden, damit sie in allen Objekten verwendet werden können. Die Funktion *percent* aus Codebeispiel 12.2 sieht in Lambda-Schreibweise wie folgt aus:

```
globals.percent = (a, b) => b * a / 100
```

Der Aufruf und auch die Zuweisung unterscheiden sich zu den anderen Funktionen.

Richtig interessant ist die Lambda-Form jedoch erst im Zusammenspiel mit anderen Funktionen. Im folgenden Beispiel wird jede Zahl einer Liste durch deren Quadrat ersetzt. Die Ersetzung geschieht hierbei mit der eingebauten Funktion *map*, ihr wird die Funktion als Argument übergeben. In der Console sieht das dann so aus:

```
<1> => var a = [ 1,2,3,4 ]
null
<2> => a.map(x => x * x)
[ 1.000000, 4.000000, 9.000000, 16.000000 ]
```

12.3.3 Verkürzte Schreibweise von Lambda-Funktion

Sollen keine Werte als Parameter übergeben werden, kann auch die verkürzte Schreibweise der Lambda-Funktion genutzt werden. Dabei wird die Funktion nicht deklariert, sondern direkt zugewiesen:

```
vars.quadr = {{ 2 * 2 }}
```

Da dieser Form von Funktionen keine Werte übergeben werden, ist hier umso wichtiger, dass sie auf Konstanten, Attribute und andere Funktionen Zugriff hat.

```
vars.rand = {{ random() + 2 }}
```

Eine Beispielfunktion, die mehrere integrierte Funktionen nutzt, ist die folgende:

```
vars.host_state = {{ get_host(macro("$host.name$")).state }}
```

In diesem Zusammenhang spielt die Funktion *macro* ein gesonderte Rolle, da sie es nicht nur gestattet, auf andere Attribute oder Custom Variables zuzugreifen, sondern auch eine Makrosubstitution über den entsprechenden Kontext vorzunehmen.

Wird sie in einem Service verwendet, setzt diese Funktion in verkürzter Lambda-Form die Custom Variable *host_state* auf den Status des dem Service zugehörigen Hosts. Erst wird *macro* benutzt, um das Makro *host.name* auszuwerten. Davon wird mit *get_host* das Hostobjekt ermittelt, von dem wiederum mit *state* der aktuelle Status ausgelesen wird.

Das folgende Beispiel zeigt auch sehr schön, dass die verkürzte Lambda-Funktion zur Laufzeit ausgewertet wird. Es ist dem Abschnitt *Advance Topics*[2] der Onlinedokumentation entlehnt.

Sei *backup* eine definierte Zeitspanne für die Uhrzeiten zwischen zwei und drei Uhr nachts am Wochenende:

```
object TimePeriod "backup" {
  ranges = {
    saturday = "02:00-03:00"
    sunday = "02:00-03:00"
  }
}
```

Codebeispiel 12.3: Timeperiod für Backups

Dann werden im folgenden Beispiel die Schwellenwerte für den gemittelten Wert über eine Minute bei *load* auf 20 und 40 gesetzt, sofern der Check innerhalb dieser Zeitspanne ausgeführt wird, andernfalls auf 5 und 10.

```
template Host "linux-host" {
  ...
  vars.load_wload1 = {{
    if (get_time_period("backup").is_inside) {
      return 20
    } else { return 5 }
  }}

  vars.load_cload1 = {{
    if (get_time_period("backup").is_inside) {
      return 40
    } else { return 10 }
  }}
}
```

Codebeispiel 12.4: Dynamisch anpassbare Schwellenwerte

[2] https://icinga.com/docs/icinga-2/latest/doc/08-advanced-topics

In diesem Beispiel wurden demnach zeitabhängige Schwellenwerte realisiert. Da die Auswertung zur Laufzeit erfolgt, ist zur Änderung der Schwellenwerte kein Neustart oder Reload erforderlich!

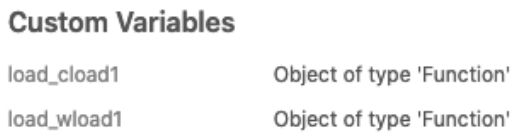

Abbildung 12-1: Anzeige von Custom Variables vom Type *function*

Einen so ermittelten Wert erkennt man immer daran, dass in Icinga Web 2 kein absoluter Wert angezeigt wird. Stattdessen erscheint der Hinweis, dass es sich um ein Objekt vom Typ *function* handelt.

Gleiches gilt für die Anzeige für solche Attribute oder Custom Variables in der Console oder in der Ausgabe von `icinga2 object list`.

12.4 Gültigkeitsbereiche

Icinga kennt für Variablen drei Gültigkeitsbereiche (engl. Scope), in aufsteigender Reihenfolge sind das die folgenden:

- lokaler Scope
- this Scope
- globaler Scope

Das bedeutet, zuerst wird im lokalen Scope gesucht, wird dort die Variable nicht gefunden, wird zunächst in this geschaut, abschließend im globalen Scope.

Lokale Variablen werden mit dem Schlüsselwort *var* deklariert. Sie gelten ausschließlich in der zugehörigen Funktion, im Objekt oder in der *apply*-Regel.

```
function sqr(x) {
  var res = x * x
  return res
}
```

Erfolgt die Deklaration in einem Ojekt oder einer *apply*-Regel ohne *var*, ist die Variable dem Scope *this* zugeordnet. Sie wird also in einem Icinga-Objekt als dessen Attribut interpretiert.

```
object Host NodeName {
  import "generic-host"
  this.check_interval = 5m
}
```

Variablen lassen sich auch explizit in einem Bereich ansprechen. Dabei wird der Variablen gemäß dem Bereich ein *locals*, *globals* oder *this* mit einem Punkt abgetrennt vorangestellt.

Anders ist der *this*-Bereich für eine Funktion auf das Objekt beschränkt, das zum Aufruf der Funktion verwendet wurde. Dabei ist hier zu beachten, dass es sich nicht notwendigerweise um ein Icinga-Objekt wie Host oder Service handeln muss.

```
obj = {
  function init(word) {
    h_word = word
  }
}
```

Initialisiert man nun das Objekt *obj*, gehört *h_word* zu *obj*:

```
<1> => obj.init("Moin!")
<2> => obj.h_word
"Moin!"
```

Standardmäßig haben Funktionen, Objekte und *apply*-Regeln keinen Zugriff auf Variablen (mit Ausnahme globaler), die außerhalb ihres Bereichs deklariert sind. Mit einer »Übergabe« mittels *use* kann dies explizit erlaubt werden.

```
template CheckCommand "by_ssh_base" {
  import "by_ssh"
  vars.by_ssh_plugindir = PluginDir
}

var cmdlist = [ "load", "procs", "disk", "ntp_time" ]

for (cmd in cmdlist) {
  object CheckCommand "by_ssh_" + cmd use(cmd) {
    import cmd
    vars.by_ssh_arguments = arguments
    arguments = null
    vars.by_ssh_command = "$by_ssh_plugindir$/check_" + cmd
    vars.by_ssh_plugindir = PluginDir
    import "by_ssh_base"
  }
}
```

Codebeispiel 12.5: Iterative Definition von Check Commands

Damit ist es möglich, iterativ mit einer »for«-Schleife Objekte zu erzeugen, hier einige Check Commands, die ein bestimmtes Plugin via SSH auf einem entfernten Host ausführen. Dabei wird jeweils auf einem bereits exis-

tierenden Check Command *cmd* aufgebaut und dessen Parametrisierung übernommen.

Es ist zwar nicht offensichtlich, aber es ist möglich, Funktionen zu definieren, die eine Funktion zurückgeben und damit ebenfalls erst zur Laufzeit ausgewertet werden. Damit ist die verkürzte Lambda-Funktion ersetzbar und unser Code wird übersichtlicher. Das Codebeispiel 12.4 auf Seite 272 verkürzt sich durch den Aufruf einer Funktion wie folgt:

```
template Host "linux-host" {
  ...
  vars.load_wload1 = dynamic_threshold("backup", 20, 5)
  vars.load_cload1 = dynamic_threshold("backup", 40, 10)
}
```

Codebeispiel 12.6: Dynamische Schwellenwerte mittels Funktion berechnet

Die nun noch benötigte Funktion *dynamic_threshold* gibt somit selbst eine Funktion zurück, die wiederum erst zur Laufzeit ausgewertet wird:

```
globals.dynamic_threshold = function(timeperiod, ivalue, ovalue) {
  return function() use (timeperiod, ivalue, ovalue) {
    if (get_time_period(timeperiod).is_inside) {
     return ivalue
    } else {
     return ovalue
    }
  }
}
```

Codebeispiel 12.7: Funktion zur Berechnung dynamischer Schwellenwerte

Abschließend seien hier noch Namensräume (engl. Namespaces) vorgestellt. Mit unterschiedlichen Namensräumen ist es möglich, Variablen, Objekte und Funktionen gegeneinander abzuschotten. Somit ist die Vergabe ein und desselben Namens möglich.

```
namespace utils {
  counter = 5
  function sqr(x) {
    return x * x
  }
}
```

Der Aufruf wird mittels des entsprechenden Namensraums als Präfix realisiert.

```
<1> => utils.counter
5.000000
<2> => utils.sqr(2)
4.000000
```

Im Konfigurationscode selbst kann auch der gesamte Namensraum geladen werden oder, anders ausgedrückt, dem globalen hinzugefügt werden, dann darf natürlich wiederum keine doppelte Vergabe von Namen stattgefunden haben:

```
using utils
var qr = sqr(3)
```