Karel de Grote Hogeschool

The Lab - Automation of Complex IT Infrastructures

Made by

Dries De Houwer

Zehra Dogan

Arno Heyvaert

Lenny Van de Winkel

1 Summary

2 Project description

In this project, we are going to review and study the different methodologies of and technologies of Infrastructure Management, Configuration Management and Automation as well as check out all the different tools that are related to this subject.

2.1 Purpose of this project

The landscape of Infrastructure Automation, Management and generally Infrastructure-as-Code and Configuration Management is evolving rapidly. Many new technologies are coming to the market that are making life easier for engineers. Many companies still work with the classic way (purely declarative Infrastructure-as-Code) of deployment. One needs to create a ticket when changes need to be made or when a test environment needs to be set up.

We are going to look at how this can be done better, how the lives of the different engineers can be made easier, how much effort and money it would take, and whether it is worth it. What are the advantages and disadvantages of the different ways and does it have a beneficial effect? What tools are available to deal with this? With several Proof of Concepts, we intend to prove the practical application of our findings in realistic use-cases.

2.2 Corporate context

Naturally, most companies are focused on making a profit: as high a revenue and as low a cost as possible. If their DevOps/Cloud/... Engineer(s) have to spend a lot of time managing the infrastructure and still deal with the trivial tasks like creating virtual machines through ticketing systems, it means that they have less time for the more important tasks and this in turn costs a company more money.

By using the latest methods to deal with this, for example with complete automation through event-driven infrastructure management, each team within the company can move forward without depending on another team for trivial matters. A developer can have a test environment set up in no time, without manual intervention from the DevOps Engineer. The developer can work faster and more smoothly, making deadlines easier to meet, customers happy and willing to pay well. Everyone's time can be put to good use and for important matters only, which means tasks will take less time which equals to less costs and more revenue in less time.

3 Provisioning vs Configuration Management

While it might look like provision and configuration management are one and the same and are often used simultaneously when deploying and managing infrastructure, there definitely is a difference between them.

A big reason for this misconception is that a lot of tools are often suitable to do both of these tasks. While provisioning is used to create the resources in your infrastructure, configuration management is about customising these resources. Provisioning usually happens before the configuration of resources.

4 Automation tools under the hood

To obtain a deeper understanding of the tools we will later investigate, we need knowledge of certain concepts, more specifically about: declarative programming, imperative programming and event driven infrastructure.

4.1 Declarative

In declarative programming, the problem to be solved is "described". The basic elements of a declarative language are therefore not commands, but descriptions of the end result. Terraform, an Infrastructure-as-Code tool, is the perfect example here within infrastructure management. Objects are declared in Terraform HCL (similar to YAML) and thus created.

4.2 Imperative

In imperative programming, statements are used to change the state of a program. In other words, with each statement something happens. Imperative programming languages describe how something happens, step by step. A perfect example of this within infrastructure management is a bash script, which sequentially executes CLI commands to get an application or infrastructure to a certain level. A big problem with this type of automation is that large infrastructures become extremely hard to manage.

4.3 Event-driven

The modern way to manage your infrastructure is event-driven. This means setting up your infrastructure management to quickly react to events that happen and to automatically take action. This method allows for quick and automatic intervention in the case of a crash, for developers to prompt for extra infrastructure to be deployed automatically without the intervention of an engineer (for example when they want to set up a test environment), for automatic scaling incase of heavy loads, you name it. Generally, this saves the engineer a lot of time manually managing infrastructure. You can define a big set of rules to account for every plausible and less-plausible scenario so that manual intervention is rarely required. In essence, an Engineer only has to set up the base infrastructure and the rest will take care of itself - manual intervention will only be necessary in cases of absolute emergency.

5 Types of Architecture

There are a couple of things that we need to consider when choosing our tools. The architecture of the tool is one of those things. The 2 most important things that we need to consider when choosing our tool based on architecture are pull or push based and agentless or agent-based architecture.

5.1 Pull Based

In a pull based architecture, there's a main server that stores a certain configuration. So-called "agents" can then connect to the main server and "pull" their configuration. This means the main server acts passively, and it's the agents' responsibility to download their configuration from the main server and carry out the necessary changes.

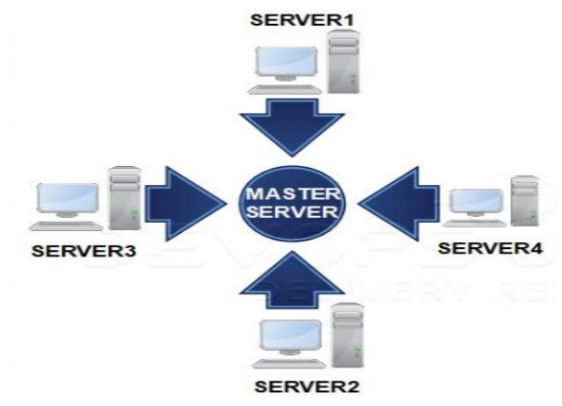


Figure 1: Pull based, servers pull from the Main

5.2 Push Based

In a push based architecture, the main server that we spoke off earlier will **actively** "push" (upload) configuration changes to all agents that are connected. In essence, the main server actively manages its underlying servers (the "agents") rather than waiting for an agent to pull the configuration by themselves.

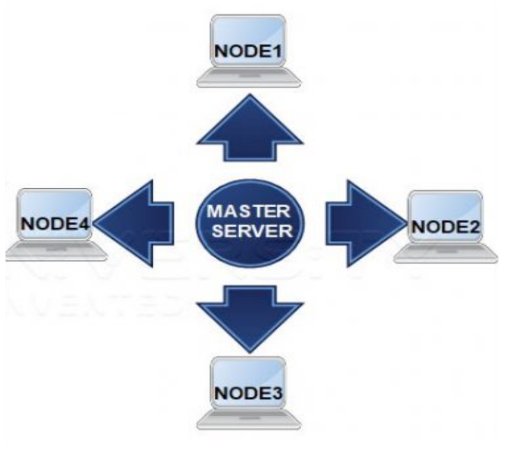


Figure 2: Push based, Main server pushes configuration to the servers

5.3 Agent-based architecture

Agent-based architecture uses a piece of software that needs to be installed on the device. This piece of software is called an agent. The agent communicates with the main server and executes certain tasks. The agent also collects data and sends it back to the main server.
Some advantages of agent-based architecture are:

- Lesser bandwidth required: data is collected locally, processed and the results are pushed to the main server.
- Security: agentless has a single vulnerable point. The main server.

5.4 Agentless architecture

Agentless design uses already installed software on the computer and doesn't depend on its own piece of management software. An example of an automation tool that uses agentless architecture is Ansible. Ansible uses SSH or Windows Remote management. Both of these tools are natively installed on all Linux and Windows based servers.
Some advantages of agentless architecture are:

- Easier management: no logging in and out manually, no manual configuration of the agent on the node.
- Reduced performance cost: agents need to run in the background.
- Reduced network chatter: no pinging between main and node servers.

6 Types of infrastructure

6.1 Immutable vs mutable

Immutable infrastructure cannot be changed after it has been deployed ("state of being unchangeable"). Should you need to make any changes, you'd need to completely re-deploy the infrastructure from scratch. Mutable infrastructure is the exact opposite. Mutable infrastructure can be changed and adapted freely after deployment.

There are a few pros of immutable infrastructure versus mutable infrastructure:

- **Predictability**: all the servers always remain the exact same.
- **Easy roll back**: previous versions remain unaffected, so rolling back is easy.
- Great for the cloud.
- Considering every server version is completely independent of all other versions, all versions are discrete and you won't have two versions running at the same time.

There are a few obvious cons, too:

- You can't modify existing servers. If there's a problem, you need to completely redeploy the server(s). Same with upgrading or changing hardware.
- Data storage needs to be external instead of local.

Every type has its use cases, however for event-driven automation we definitely need mutable infrastructure.

7 Well known tools

7.1 Ansible

7.1.1 What is Ansible?

Ansible is an open source software provisioning, configuration management and deployment tool that enables users to configure Unix and Windows systems. All of this can be achieved by using the human readable declarative language that Ansible uses.

7.1.2 How does it work?

Ansible is written in Python. Users can use modules to interact with their local system or remote system(s). It's possible to make your own modules using Python, Powershell, Bash or Ruby. Ansible is agentless and push based. It makes use of SSH or Windows Remote Management to execute it's playbooks on remote nodes.

Ansible makes use of playbooks and inventories. Playbooks use the YAML format and contain one or more plays. A play executes one or more tasks.
An inventory defines all hosts and groups of hosts.

An example of a simple inventory can be found below. This inventory contains 2 groups (webservers & dbservers) and the web server group consists of 2 systems and the dbservers group contains 3 systems.

```
[webservers]
foo.example.com
bar.example.com

[dbservers]
one.example.com
two.example.com
three.example.com
```

An example of a simple playbook can be found below. This playbook contains 2 plays. Both plays use an elevated user (root). The first play updates Apache and starts the service on all web servers. The second play updates postgresql and starts the service on all database servers.

```
---
  - name: play1
    hosts: webservers
    remote_user: root
    tasks:
      - name: ensure apache is at the latest version
        ansible.builtin.yum:
          name: httpd
          state: latest
      - name: ensure apache is running
        ansible.builtin.service:
          name: httpd
          state: started

  # Play 2: update and start postgresql on all dbservers
  - name: play2
    hosts: dbservers
    become: yes
    become_user: root
    tasks:
      - name: ensure postgresql is at the latest version
        ansible.builtin.yum:
          name: postgressql
          state: latest
      - name: ensure postgresql is running
        ansible.builtin.service:
          name: postgresql
          state: started
```

7.2 Puppet

7.2.1 What is Puppet?

Puppet is an open-source configuration management solution, which is built with Ruby and offers custom Domain Specific Language (DSL) and Embedded Ruby (ERB) templates to create custom Puppet language files, offering a declarative-paradigm programming approach.

There are two versions: an open-source and an enterprise version.

7.2.2 How does it work?

Puppet uses a declarative approach and has a pull based architecture.

7.2.2.1 Components

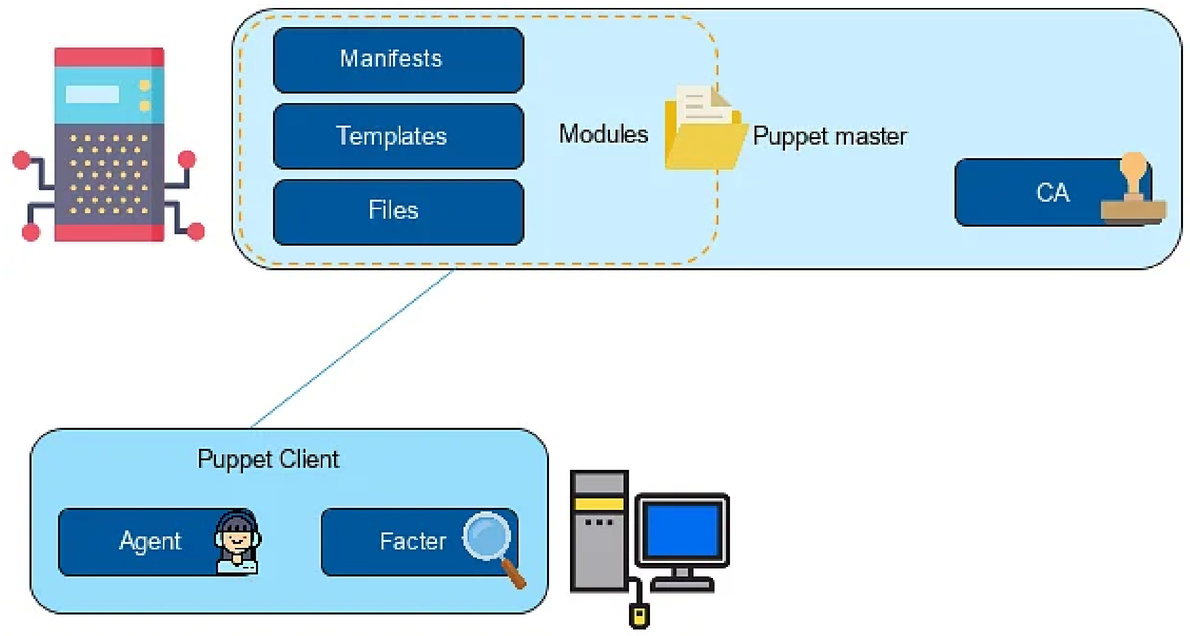


Figure 3: Components of Puppet

The Main server contains:

- Manifests : Actual code for configuring the clients
- Templates: Combines code and data together to render a final document
- Files: Static content that can be downloaded by clients
- Modules are a collection of manifests, templates, and files
- Certificate authority: allows the Main server to sign the certificates sent by the client

The Client server contains:

- Agents: Interacts with Main server to ensure certificates are updated
- Facter: Collects current state of the client (facts) and sends it to Main server

7.2.2.2 Writing Manifests

Manifests are written in Ruby and need a .pp extension.

Here is an example of how we write manifests:

```
Resourcetype { 'title':
   Attribute_name1 => attribute_value1,
   Attribute_name2 => attribute_value2
   …
}
```

Resourcetypes can either be a package, a file or a service.

7.3 Terraform

7.3.1 What is Terraform?

Terraform is an open-source Infrastructure as Code (IaC) tool made by Hashicorp. It uses the configuration files written in Hashicorp Configuration Language (HCL - based on YAML) to configure, change or remove an infrastructure. Terraform can be used for multiple cloud providers like Google Cloud Platform, Azure and Amazon Web Services.

7.3.2 How does it work?

Terraform uses blocks to describe infrastructure components. Inside these blocks you can use different identifiers, depending on the resource you want to create, to configure that resource. Some examples of resources you can create using Terraform are virtual machines, databases and networking rules.

Below is an example of a configuration of a MySQL database in Microsoft Azure.

```
resource "azurerm_mysql_database" "database" {
  name= "mysqldb"
  resource_group_name = azurerm_resource_group.databaseGroup.name
  server_name = azurerm_mysql_server.dbServer.name
  charset = "utf8"
  collation = "utf8_general_ci"
}
```

The Terraform workflow consists of three stages, namely the write stage, the plan stage and the apply stage.

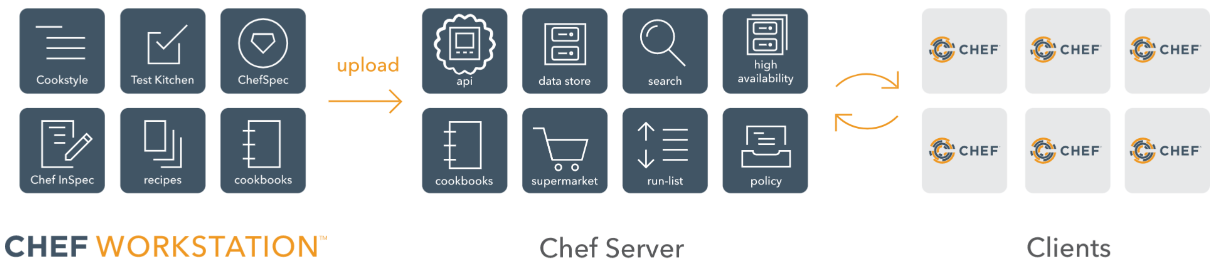
The **write** stage is self explanatory. In this stage you write your blocks in your configuration files.

During the **planning** stage, Terraform takes your configuration files and compares them with the current status of all resources. It will show all the resources that will be created, deleted or changed when applying the current configuration. Terraform will figure out which resources depend on each other and it also figures out the right order of deployment in the planning stage. The changes aren't deployed automatically, Terraform will ask for permission before applying these changes.

The last stage is the **apply** stage. In this stage Terraform will deploy your approved plan from the planning phase and will notify you when the deployment is complete.

7.4 Chef Infra

7.4.1 What is Chef Infra?



Chef Infra is a part of a collection of tools that serve different purposes within infrastructure automation, developed by Progress Software. For the purposes of this paper, we're only going to take a look at one of those tools, **Chef Infra**: a configuration management tool for defining Infrastructure as Code (IaC).

7.4.2 How does it work?

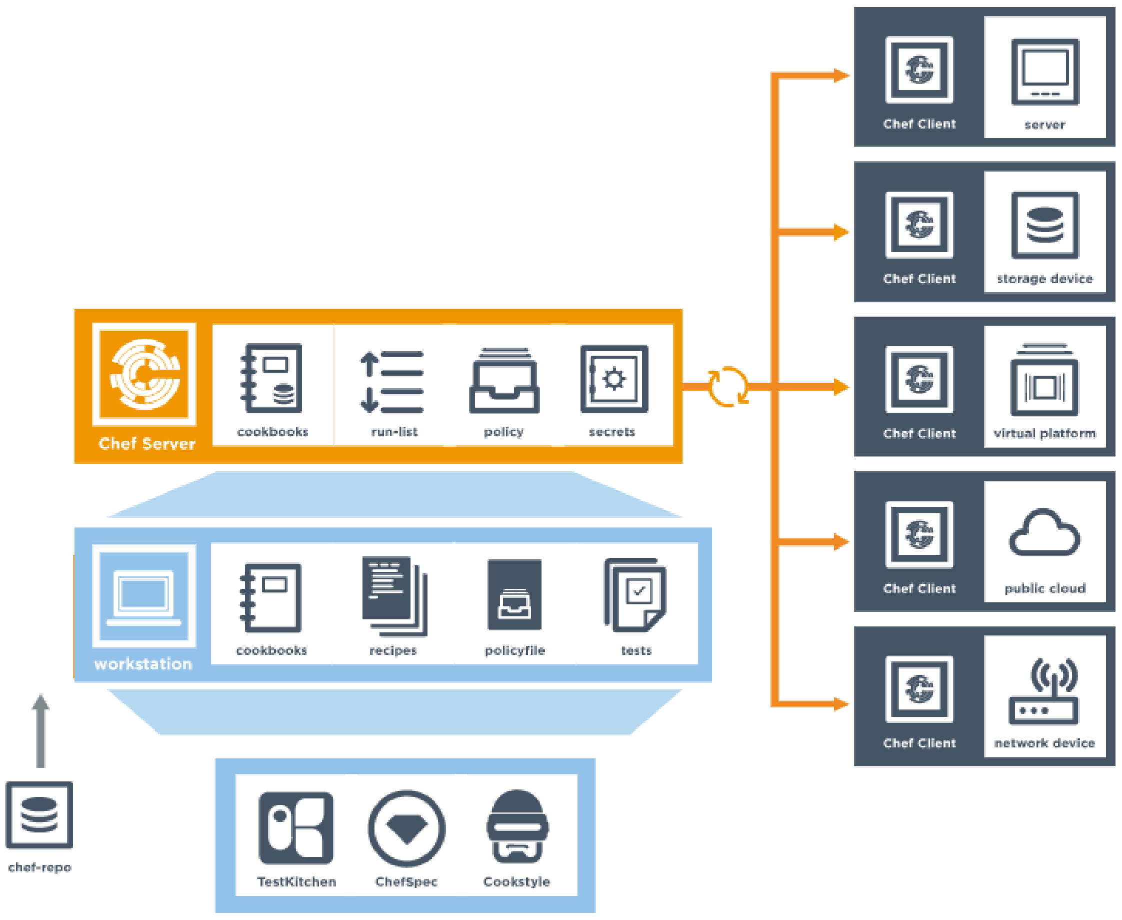
Chef Infra has two different installations, a Chef Infra Server and Chef Infra Clients. It uses the Chef Infra Language, which is a language specifically designed for Chef Infra, written in Ruby.

Chef uses **cookbooks** for configuration. A cookbook is essentially a collection of recipes (much like an actual cookbook) that defines how infrastructure is deployed.

Chef Infra has a few components to it:

- First, there's the Chef Infra Server. It houses all of the configuration details for underlying nodes.
- Then there's nodes. Every node has the Chef Infra Client installed on it, which allows it to communicate with the Chef Infra Server to pull their configuration.
- The Chef Workstation is a tool that allows administrators to write, test and eventually push cookbooks to the Chef Infra Server.
- Cookbooks consist of multiple recipes (much like an actual cookbook) which, in short, defines how nodes are configured. Cookbooks are written in Ruby.

There's a lot more to Chef Infra that we will not cover in this paper, but below is a diagram about all of the different components that Chef Infra consists of and how they communicate.



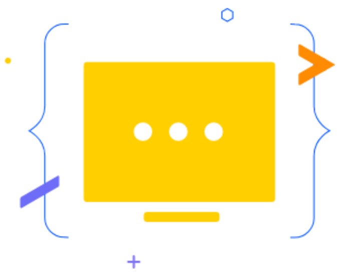
7.4.2.1 Chef Infra Server

The Chef Infra Server is a central hub where all of the cookbooks, policies et cetera are stored. Nodes can use the Chef Infra Client to "talk" with this server to retrieve configuration details. Chef Infra is pull-based, and thus the responsibility to keep up-to-date lies with the clients.

7.4.2.2 Chef Infra Client

Every node has the Chef Infra Client installed on it. It will pull configurations from the Chef Infra Server and apply it to the machine it's installed on.

7.4.2.3 Chef Workstation



Chef Workstation gives you everything you need to get started with Chef Infra and Chef InSpec — ad hoc remote execution, remote scanning, configuration tasks, cookbook creation tools as well as robust dependency and testing software — all in one package. Essentially, it allows you to create and test configurations before they get pushed to production.

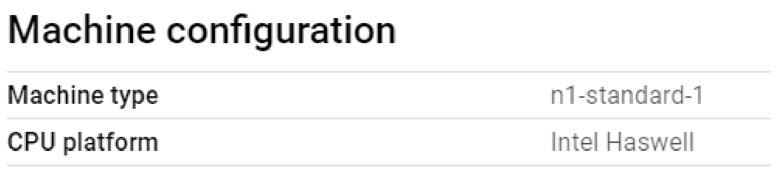
8 Tool comparison

8.1 Ansible vs Terraform

TODO: verschillen tussen Ansible en Terraform uitleggen qua infrastructuur deployment. Vergelijking van beide tools in de praktijk (config of iets anders, …)

We will use Terraform in this small use case to spin up a MySQL database and a compute instance on the Google Cloud Platform. We will then deploy a simple .NET web app on this compute instance.

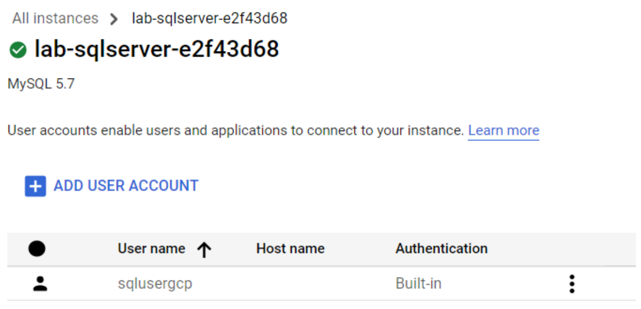
For this small use case we have used terraform to quickly spin up a mysql database and a virtual machine on which we use docker to host our .NET web application. As you can see in the picture below , our vm machine type currently is n1-standard-1.



Machine configuration

Machine type	n1-standard-1
CPU platform	Intel Haswell

When we take a look at our MySQL database we can see in the users section that our database currently has one user that we created when creating the database.



All instances > lab-sqlserver-e2f43d68

lab-sqlserver-e2f43d68

MySQL 5.7

User accounts enable users and applications to connect to your instance. Learn more

	User name	Host name	Authentication
	sqlusergcp		Built-in

To show how Terraform deals with automation we are now going to upgrade the machine tier of our webapp virtual machine to the n2-standard-2 tier and add an extra user to our MySQL database.

The first to achieve this is to change our terraform configuration file. As you can see on the 2 pictures below we change the machine_type identifier for our webapp resource to n2-standard-2 and create a new mysql user named newUser.

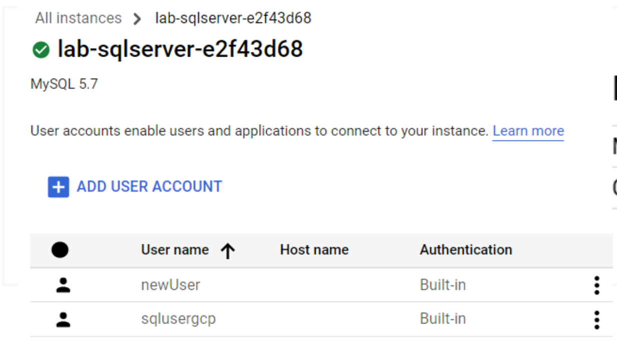
```
resource "google_compute_instance" "webApp" {
  name             = "terraform-dotnetapp-n2"
  machine_type     = n2-standard-2
  zone           = var.zone

  ...
}

resource "google_sql_user" "newUser" {
  name     = "newUser"
  instance = google_sql_database_instance.gcp-cloud-sql.name
  password = var.gcp_sql_newUser_pass
}
```

The next step is to execute the terraform plan command. Terraform will now compare the deployed state to the state set in the configuration files and display how many resources will be created,changed and destroyed. In our case the terraform plan will inform us that it will create two new resources and destroy one when we decide to deploy this plan.

```
~$:terraform plan
...

Terraform will perform the following actions:

  # google_compute_instance.webapp will be created
  + resource "google_compute_instance" "terraform" {
      + machine_type            = "n2-standard-2"
      ...
}

  # google_compute_instance.webApp will be destroyed
  - resource "google_compute_instance" "webApp"  {
      - machine_type            = "n1-standard-1" -> null
      ...
}

 # google_sql_user.newUser will be created
  + resource "google_sql_user" "newUser" {
      + host     = (known after apply)
      + id       = (known after apply)
      + instance = "lab-sqlserver-e2f43d68"
      + name     = "newUser"
      + password = (sensitive value)
      + project  = (known after apply)
    }
```

```
Plan: 2 to add, 0 to change, 1 to destroy.
```

The final step is to now deploy these changes with the terraform apply command. Once the deployment of the plan has finished we can go check if our changes have been successfully made and as we can see on the 2 pictures below there was indeed another user added to our database and the machine type of our compute instance is now n2-standard-2.

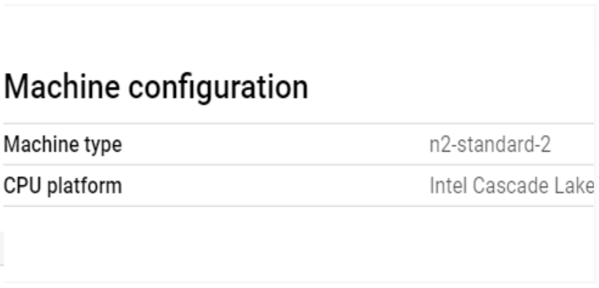

All instances > lab-sqlserver-e2f43d68

lab-sqlserver-e2f43d68

MySQL 5.7

User accounts enable users and applications to connect to your instance. Learn more

●	User name ↑	Host name	Authentication
	newUser		Built-in
	sqlusergcp		Built-in


Machine configuration

Machine type	n2-standard-2
CPU platform	Intel Cascade Lake

8.2 Ansible vs Chef vs Puppet

TODO: praktisch uitwerken, daarna tools vergelijken met elkaar, …

9 Event-Driven Automation

9.1 What is event-driven?

Event-driven automation uses **events** to trigger processes and actions without any manual intervention. Events are in essence any change to a system or an environment: a good example being a certain RAM threshold being reached, or a file being edited. Once an event occurs, an automation tool (like StackStorm) can take action automatically. A frequently used use-case for this is auto-scaling: once one or more data points (CPU, RAM, …) reach a certain threshold, extra servers are automatically deployed (horizontal autoscaling).

In the past, automation was mostly done with date and time scheduling. While this remains a good way to automate some tasks such as end-of-the-week report generation, a lot of tasks can't be done reliably purely with date and time.

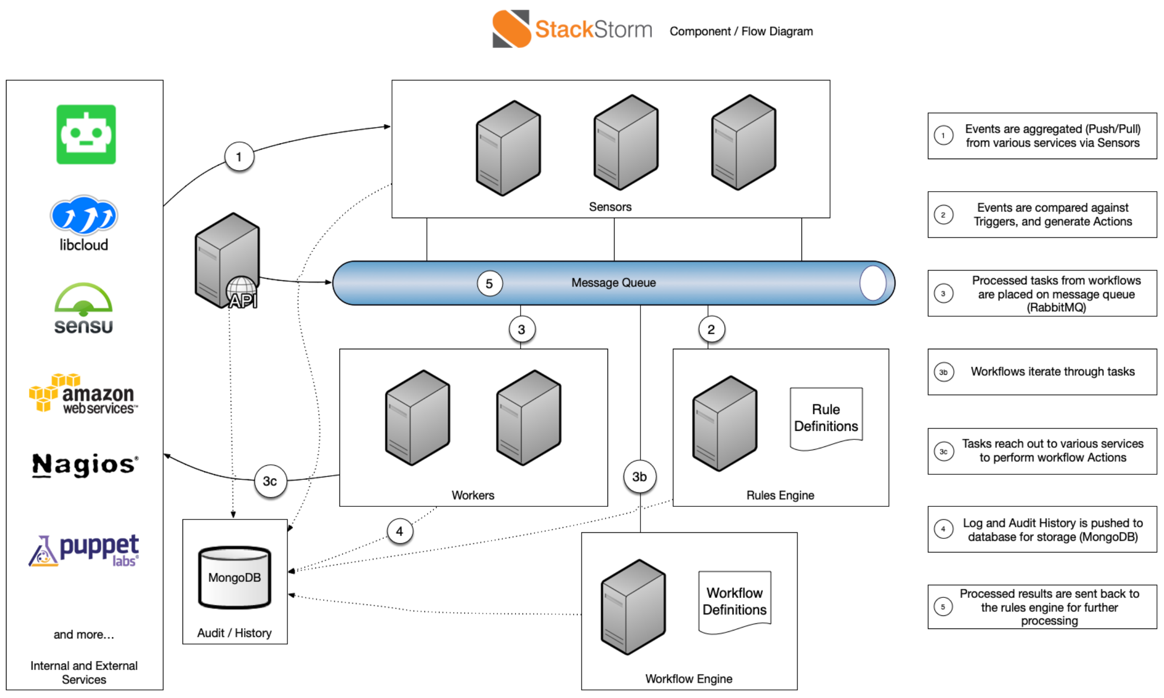
9.2 Advantages

Without the use of events, when a status changes, we usually get a notification that someone has to check and take action manually. Thanks to event-driven automation there is no need to do this. If a certain status changes an event gets triggered, and the necessary configuration will be executed automatically. We are eliminating unnecessary work.

10 StackStorm

StackStorm is a platform used for integration and automation. It connects your (existing) infrastructure with your application environment so that automation is made easy. Essentially, StackStorm takes actions in response to events.

10.1 How does StackStorm work?


StackStorm Component / Flow Diagram
libcloud
sensu
amazon web services
Nagios
puppet labs
and more...
Internal and External Services
API
Sensors
Message Queue
Workers
Rules Engine
Rule Definitions
MongoDB
Audit / History
Workflow Definitions
Workflow Engine
1
2
3
3b
3c
4
5
Events are aggregated (Push/Pull) from various services via Sensors
Events are compared against Triggers, and generate Actions
Processed tasks from workflows are placed on message queue (RabbitMQ)
Workflows iterate through tasks
Tasks reach out to various services to perform workflow Actions
Log and Audit History is pushed to database for storage (MongoDB)
Processed results are sent back to the rules engine for further processing

StackStorm plugs into an environment via adapters which contain sensors and actions.

StackStorm is very complex and we'll only cover the fundamentals in this paper. If you'd like to know more about StackStorm, feel free to take a look at the official documentation.

10.1.1 Sensor

A **Sensor** is a Python plugin that receives or watches for events (inbound and outbound respectively). If a Sensor processes an event, a StackStorm **trigger** is activated. Every sensor runs as a separate process on the system.

Creating a sensor requires two things: a Python file and a YAML metadata file that defines the sensor itself. A minimal example:

```yaml
---
  class_name: "SampleSensor"
  entry_point: "sample_sensor.py"
  description: "Sample sensor that emits triggers."
  trigger_types:
    -
      name: "event"
      description: "An example trigger."
      payload_schema:
        type: "object"
        properties:
          executed_at:
            type: "string"
            format: "date-time"
            default: "2014-07-30 05:04:24.578325"
```

And this is the corresponding Python file:

```python
# Copyright 2020 The StackStorm Authors.
# Copyright 2019 Extreme Networks, Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from st2reactor.sensor.base import Sensor


class SampleSensor(Sensor):
    """
    * self.sensor_service
        - provides utilities like
            - get_logger() - returns logger instance specific to this sensor.
            - dispatch() for dispatching triggers into the system.
    * self._config
        - contains parsed configuration that was specified as
          config.yaml in the pack.
    """

    def setup(self):
        # Setup stuff goes here. For example, you might establish connections
        # to external system once and reuse it. This is called only once by the system.
        pass

    def run(self):
        # This is where the crux of the sensor work goes.
        # This is called once by the system.
        # (If you want to sleep for regular intervals and keep
        # interacting with your external system, you'd inherit from PollingSensor.)
        # For example, let's consider a simple flask app. You'd run the flask app here.
        # You can dispatch triggers using sensor_service like so:
        # self.sensor_service(trigger, payload, trace_tag)
        #   # You can refer to the trigger as dict
        #   # { "name": ${trigger_name}, "pack": ${trigger_pack} }
        #   # or just simply by reference as string.
        #   # i.e. dispatch(${trigger_pack}.${trigger_name}, payload)
        #   # E.g.: dispatch('examples.foo_sensor', {'k1': 'stuff', 'k2': 'foo'})
        #   # trace_tag is a tag you would like to associate with the dispatched
TriggerInstance
        #   # Typically the trace_tag is unique and a reference to an external event.
        pass

    def cleanup(self):
        # This is called when the st2 system goes down. You can perform cleanup
operations like
        # closing the connections to external system here.
        pass

    def add_trigger(self, trigger):
        # This method is called when trigger is created
        pass

    def update_trigger(self, trigger):
        # This method is called when trigger is updated
        pass

    def remove_trigger(self, trigger):
        # This method is called when trigger is deleted
        pass
```

Keep in mind that this example is a very minimal example of what a Sensor can do. There's a load of examples available on the StackStorm documentation, which involve a much more complex Sensor creation. You can instead of passively waiting for Sensor input, actively poll an external system at regular intervals:

```
from st2reactor.sensor.base import PollingSensor

class SamplePollingSensor(PollingSensor):
    """
    * self.sensor_service
        - provides utilities like
```

```
            get_logger() for writing to logs.
            dispatch() for dispatching triggers into the system.
    * self._config
        - contains configuration that was specified as
          config.yaml in the pack.
    * self._poll_interval
        - indicates the interval between two successive poll() calls.
    """

    def setup(self):
        # Setup stuff goes here. For example, you might establish connections
        # to external system once and reuse it. This is called only once by the system.
        pass

    def poll(self):
        # This is where the crux of the sensor work goes.
        # This is called every self._poll_interval.
        # For example, let's assume you want to query ec2 and get
        # health information about your instances:
        #   some_data = aws_client.get('')
        #   payload = self._to_payload(some_data)
        #   # _to_triggers is something you'd write to convert the data format you have
        #   # into a standard python dictionary. This should follow the payload schema
        #   # registered for the trigger.
        #   self.sensor_service.dispatch(trigger, payload)
        #   # You can refer to the trigger as dict
        #   # { "name": ${trigger_name}, "pack": ${trigger_pack} }
        #   # or just simply by reference as string.
        #   # i.e. dispatch(${trigger_pack}.${trigger_name}, payload)
        #   # E.g.: dispatch('examples.foo_sensor', {'k1': 'stuff', 'k2': 'foo'})
        #   # trace_tag is a tag you would like to associate with the dispatched
TriggerInstance
        #   # Typically the trace_tag is unique and a reference to an external event.
        pass

    def cleanup(self):
        # This is called when the st2 system goes down. You can perform cleanup
operations like
        # closing the connections to external system here.
        pass

    def add_trigger(self, trigger):
        # This method is called when trigger is created
        pass

    def update_trigger(self, trigger):
        # This method is called when trigger is updated
        pass

    def remove_trigger(self, trigger):
        # This method is called when trigger is deleted
        pass
```

10.1.2 Triggers

Triggers are representations of external events. For example, a timer and a webhook (generic triggers) or a JIRA issue that's updated (integration triggers). The latter, integration triggers, aren't native to StackStorm and are meant to integrate with third party products.

10.1.2.1 Internal Triggers

By default, StackStorm has some internal triggers which can be used in rules. These triggers can be identified easily as they are prefixed by `st2`. A list of available triggers can be found in the documentation.

After a sensor is made, a trigger is generated in Python dict form:

```
trigger = 'pack.name'
payload = {
    'executed_at': '2014-08-01T00:00:00.000000Z'
}
trace_tag = external_event_id
```

10.1.3 Actions

Actions are self-explanatory. An action is initiated by a trigger and can do many things: generic actions (SSH, REST calls, …), integrations (Docker, Git, …) or custom user-defined actions. The metadata of these actions is written in **YAML**, which runs a script (most often Python) that executes the action.

A few examples of tasks which can be done with actions:

- Managing (starting, stopping and restarting) a service on a server.
- Creating a new cloud server.
- Start a Docker container.
- Send a notification via email.

Managing actions can be done via the CLI, you can get all the commands and their descriptions by running:

```
st2 action --help
```

```
st2 action list -h
```

A YAML metadata example of an action that sends an SMS:

```
---
name: "send_sms"
runner_type: "python-script"
description: "This sends an SMS using twilio."
enabled: true
entry_point: "send_sms.py"
parameters:
    from_number:
        type: "string"
        description: "Your twilio 'from' number in E.164 format. Example
+14151234567."
        required: true
        position: 0
    to_number:
        type: "string"
        description: "Recipient number in E.164 format. Example
+14151234567."
        required: true
        position: 1
        secret: true
    body:
        type: "string"
        description: "Body of the message."
        required: true
        position: 2
        default: "Hello {% if system.user %} {{ st2kv.system.user }} {%
else %} dude {% endif %}!"
```

10.1.3.1 Action Runners

An action runner is the execution environment for custom actions (user implemented). StackStorm comes with a series of runners by default, which each have their own purpose.

- local-shell-cmd - This is the local runner. This runner executes a Linux command on the host where StackStorm is running.
- local-shell-script - This is the local runner. Actions are implemented as scripts. They are executed on the hosts where StackStorm is running.
- remote-shell-cmd - This is a remote runner. This runner executes a Linux command on one or more remote hosts provided by the user.
- remote-shell-script - This is a remote runner. Actions are implemented as scripts. They run on one or more remote hosts provided by the user.

- python-script - This is a Python runner. Actions are implemented as Python classes with a run() method. They run locally on the same machine where StackStorm components are running. The return value from the action run() method is either a tuple of success status flag and the result object respectively or it is just the result object. For more information, please refer to the Action Runners section in the documentation.
- http-request - HTTP client which performs HTTP requests for running HTTP actions.
- action-chain - This runner supports executing simple linear work-flows. For more information, please refer to the Workflows and ActionChain documentation.
- inquirer - This runner provides the core logic of the Inquiries feature.
- Note: This runner is an implementation detail for the core.ask action, and in most cases should not be referenced in other actions.
- winrm-cmd - The WinRM command runner allows you to run the command-line interpreter (cmd) commands on Windows hosts using the WinRM protocol.
- winrm-ps-cmd - The WinRM PowerShell command runner allows you to run the PowerShell commands on Windows hosts using the WinRM protocol.
- winrm-ps-script - WinRM PowerShell script runner allows you to run PowerShell scripts on Windows hosts.
- orquesta - This runner supports executing complex work-flows. For more information, please refer to the Workflows and Orquesta documentation.

10.1.3.2 Writing Custom Actions

An action is composed of a YAML metadata file defining the action and a script file which implements the logic.

Action metadata is as follows:

- name - Name of the action.
- runner_type - The type of runner to execute the action.
- enabled - Action cannot be invoked when disabled.
- entry_point - Location of the action launch script relative to the /opt/stackstorm/packs/${pack_name}/actions/ directory.
- parameters - A dictionary of parameters and optional metadata describing type and default. The metadata is structured data following the JSON Schema specification draft 4. The common parameter types allowed are string, boolean, number (whole numbers and decimal numbers - e.g. 1.0, 1, 3.3333, etc.), object, integer (whole numbers only - 1, 1000, etc.) and array. If metadata is provided, input args are validated on action execution. Otherwise, validation is skipped.
- tags - An array with tags for this actions for the purpose of providing supplemental information

There's a series of available techniques, such as the usage of certain environment variables, parameters et cetera. This can be found in the documentation.

10.1.4 Rules

Rules define which actions are activated by what triggers by means of criteria and inputs.

As with any other StackStorm component, rules are defined in YAML with the following structure and elements:

```yaml
---
    name: "rule_name"                      # required
    pack: "examples"                       # optional
    description: "Rule description."       # optional
    enabled: true                          # required

    trigger:                               # required
        type: "trigger_type_ref"

    criteria:                              # optional
        trigger.payload_parameter_name1:
            type: "regex"
            pattern : "^value$"
        trigger.payload_parameter_name2:
            type: "iequals"
            pattern : "watchevent"

    action:                                # required
        ref: "action_ref"
        parameters:                        # optional
            foo: "bar"
            baz: "{{ trigger.payload_parameter_1 }}"
```

10.1.5 Workflows

Workflows stitch actions together into "uber-actions". You can define a series of actions with a specific order, transition conditions et cetera. Essentially this allows you to create more steps to a trigger.

In order to create a workflow, you need to choose a workflow runner and set it up according to the documentation:

- Orquesta
- ActionChain

10.1.6 Packs

Packs group integrations (triggers and actions) and automations (rules and workflows) into a single package. Packs can then be shared and downloaded. It's much like a Docker image, containing an entire pre-configured installation. There are two types of packs:

- **Integration packs**: packs that integrate StackStorm with external systems (AWS, Github, JIRA, …)
- **Automation packs**: packs that contain automation patterns, being workflows, rules and actions.

StackStorm packs are managed with `st2 pack`. You can put 'list' or 'get' behind them, to list all packages or download one respectively.

10.2 Installation

Installation of StackStorm is simple enough. They offer several options, which are explained in detail in their documentation, but are explained briefly below:

- The simplest option is the **one-line install**. This downloads and runs a bash script that will install StackStorm with the recommended components.
- **Manual installation**: perfect for those with custom needs or no internet connection.
- StackStorm is also available as a **Vagrant Virtual Image**, which allows it to run virtually.
- **Docker** also supports StackStorm and is the quickest and easiest way to use it.
- If you use **Ansible**, you can also use an Ansible Playbook to deploy StackStorm.
- A **Puppet Module** is also available.
- Lastly, you can run StackStorm in a **Kubernetes** cluster for High Availability.

11 Proof of concept

Idea: JIRA integration to automatically create a user in AD or lock a user (after for example getting fired).

12 Conclusion

13 Glossary

14 Sources

https://www.youtube.com/watch?v=lV1g9-Zta1M
https://docs.chef.io/
https://www.terraform.io/intro
StackStorm Documentation — StackStorm 3.6.0 documentation