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. We will mainly focus on an event driven tool called StackStorm, and explore the possibilities it offers us.

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 types

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

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 a description of the end result. Terraform, an Infrastructure-as-Code tool, is the perfect example of a declarative tool. Objects are declared in Terraform HCL (similar to YAML) and then 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 infrastructure will take care of itself - manual intervention will only be necessary in cases of absolute emergency.
Event-driven uses events to trigger processes and actions to achieve this. An example of this can be a RAM threshold that has been reached. The environment will automatically create a new instance to spread the load between the instances.
In the past, automation was mostly done with date and time scheduling. While this remains a good way to automate certain tasks such as end-of-the-week report generation, a lot of tasks can't be done reliably purely with date and time.

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.

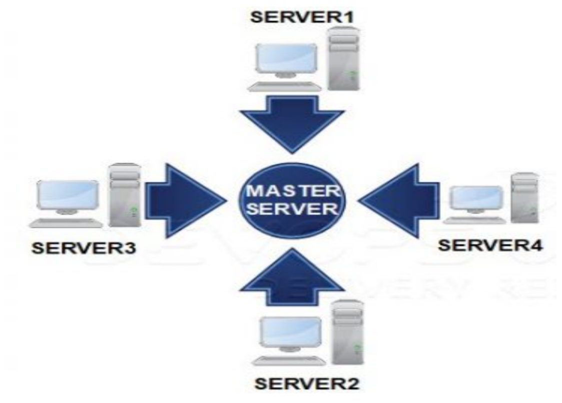


Figure 1: Pull based, servers pull from the Main

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.

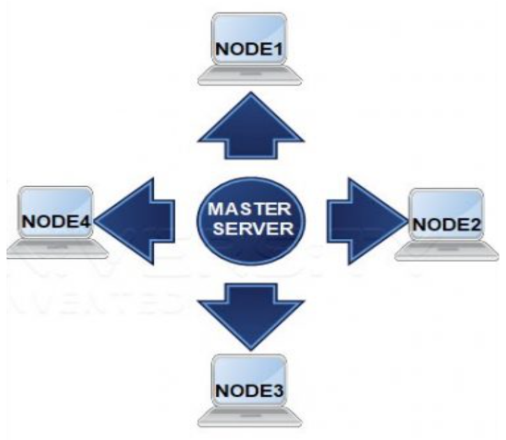


Figure 2: Push based, Main server pushes configuration to the servers

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. These tools are natively installed on all Linux and Windows based devices.
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.

An example of immutable infrastructure are containers. Once an image is built, it's unchangeable. When changes are made, a new image has to be created. Compute engine instances are mutable infrastructure. These instances can be modified, updated, … after they are created.

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 its 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.

```yaml
---
  - 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 of Puppet: 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. Puppet uses manifest files. Manifest files are written in Ruby and allow users to describe the wanted state of a device. Like Ansible, Puppet also makes use of modules. Modules have their own specific task and can be shared and reused.

7.2.2.1 Components

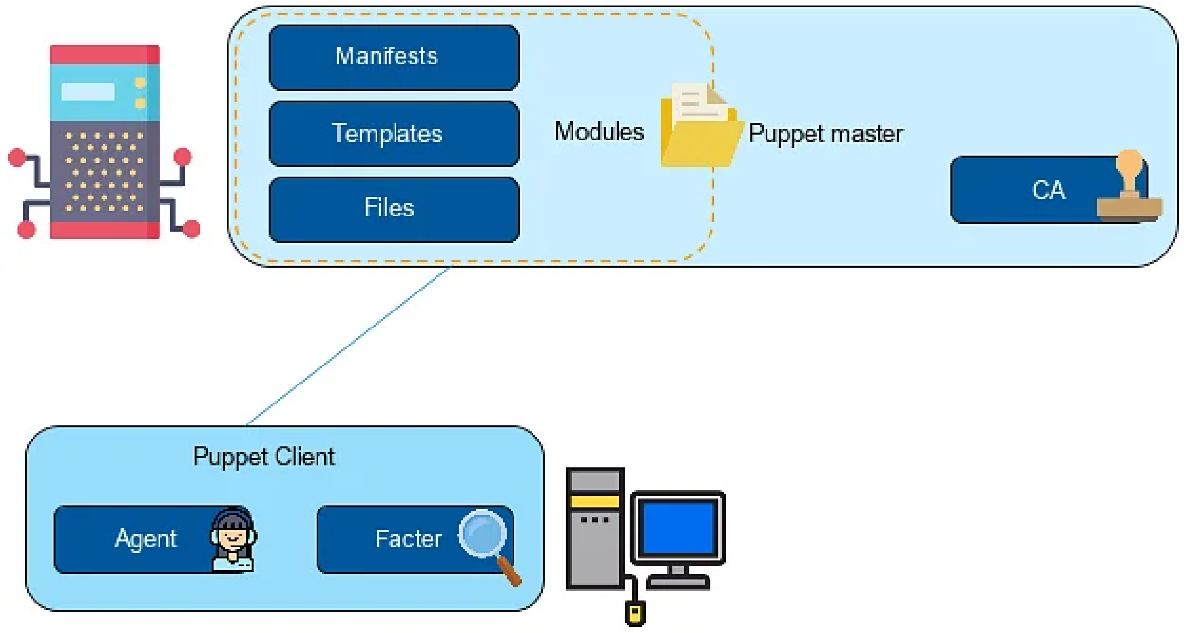


Figure 3: Components of Puppet

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.

An example of a manifest file can be found below. This example updates the instance and installs the Git package. The node block specifies that these actions need to be executed on the 'puppetagent.local' node.

```
Node 'puppetagent.local' {
  exec { 'apt-update':
    command => '/usr/bin/apt-get update -y'
  }

  package { 'git':
    require => Exec['apt-update'],
    ensure => installed,
  }
}
```

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.

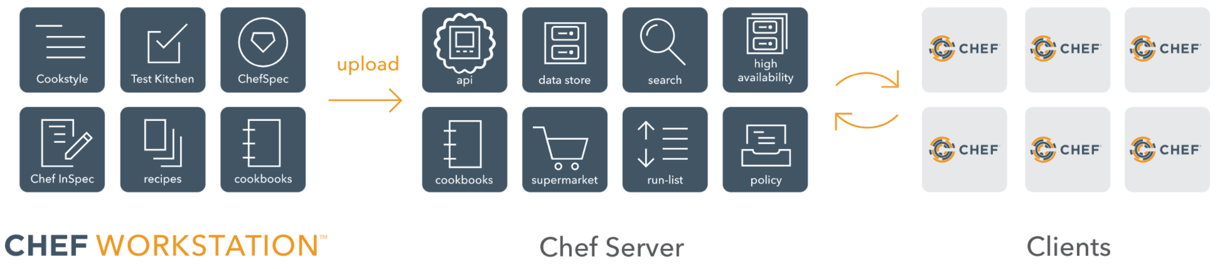
The write stage is self explanatory. In this stage you write your blocks in your configuration files.

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?

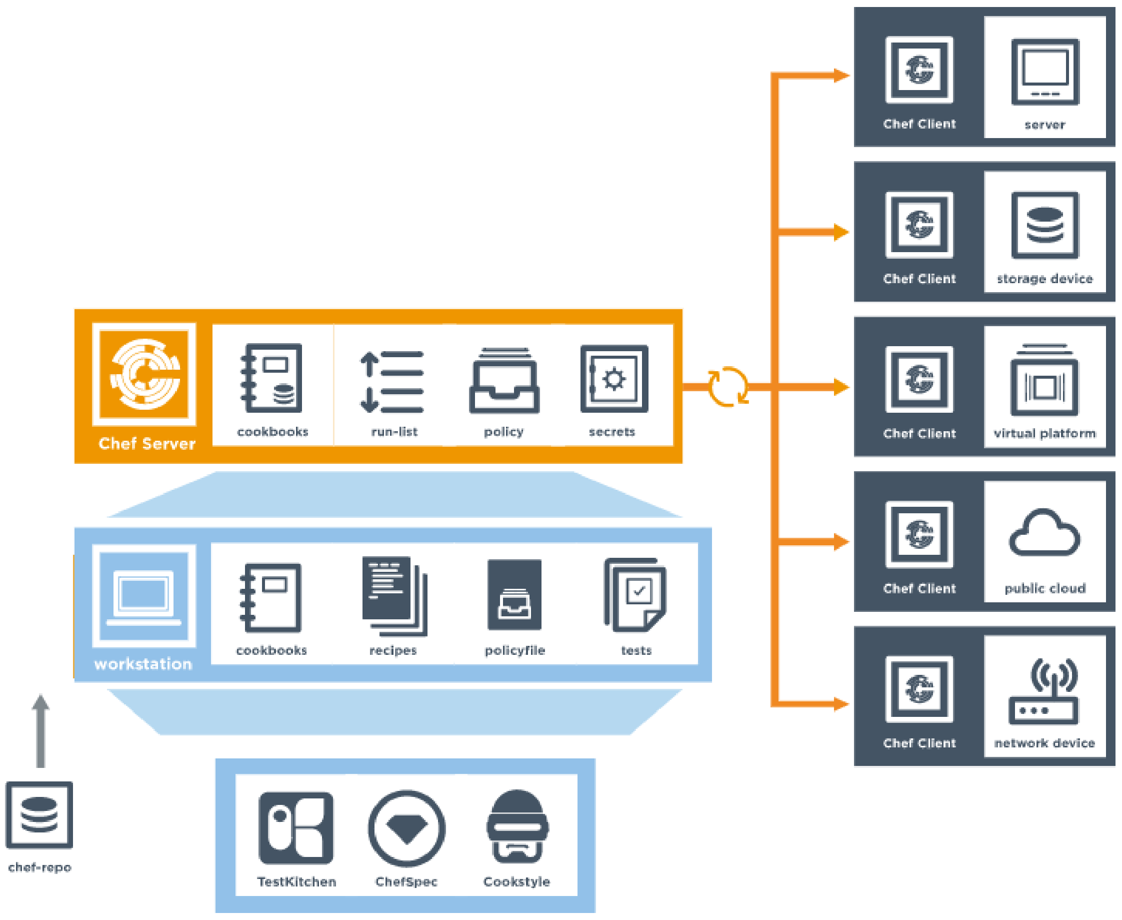
Chef Infra has two different installations, a Chef Infra Server and Chef Infra Clients. It uses the Chef Infra Language, which is a language specifically designed for Chef Infra, written in Ruby.

Chef uses cookbooks for configuration. A cookbook is essentially a collection of recipes (much like an actual cookbook) that define how software 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.



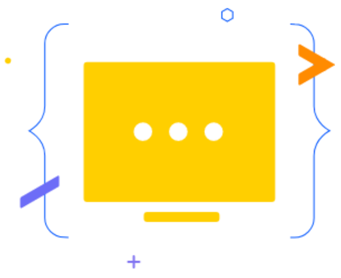
7.4.2.1 Chef Infra Server

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

While Ansible is best known for software provisioning, Ansible can also provision infrastructure. We are going to compare the infrastructure provisioning capabilities of Ansible and Terraform, by using both tools to create the same environment on GCP (Google Cloud Platform). Both environments will consist of a VPC network, a MySQL database and a compute instance. We won't be diving into the software provisioning part since we're only comparing the infrastructure part of these tools. The code for both of these tools can be found on our Git repository.

8.1.1 Declaration

One of the main differences between the declaration of both tools is how we define our resources. While Terraform uses HCL (Hashicorp Configuration Language) or JSON, Ansible uses YAML. The code block below shows the creation of a VPC with Terraform and Ansible.

```
# Terraform: VPC
resource "google_compute_network" "vpc" {
  name = "terraform-vpc"
  auto_create_subnetworks = "false"
}

# Ansible: VPC
- name: 'Create network'
      gcp_compute_network:
        name: ansible-network
        auto_create_subnetworks: true
        project: "{{ project }}"
        auth_kind: "{{ auth_kind }}"
        service_account_file: "{{ service_account_file }}"
        state: present
      register: network
```

Another example of the difference between the declaration can be found in our Git repository. While Terraform needs one code block to define a compute instance resource, Ansible needs 4 different plays (disk, address, network, instance) to create a single compute instance.

8.1.2 User friendliness

In terms of user friendliness, Terraform wins by a wide margin. The order in which resources are defined doesn't matter because Terraform will figure out the order in which to deploy resources by itself.

While it isn't best practice, all configuration in Terraform can be done in one file. Ansible on the other hand needs at least 2 files: one for creating and one for deleting the resources. This makes consistency when changing existing resources using Ansible very important, as both files need to be changed whereas Terraform only needs changes in 1 file.

8.2 Ansible vs Chef vs Puppet

8.2.1 Architecture

While Ansible is based on an agentless configuration, both Chef and Puppet use an agent-based configuration. Ansible can both use pull and push based configuration. Chef and Puppet, on the other hand, use a pull based configuration.

8.2.2 Declaration

All configuration in Ansible is done via YAML. Configuration in Chef is done using Ruby. Puppet uses Ruby and Puppet DSL (Domain Specific Language) to configure all resources. YAML is easy to read, this ensures that Ansible configuration files are easy to read and understand. This makes Ansible the easiest to learn when comparing it to Chef and Puppet.
We have made a practical comparison between all tools by deploying the same application using these tools. The code can be found on our Git.

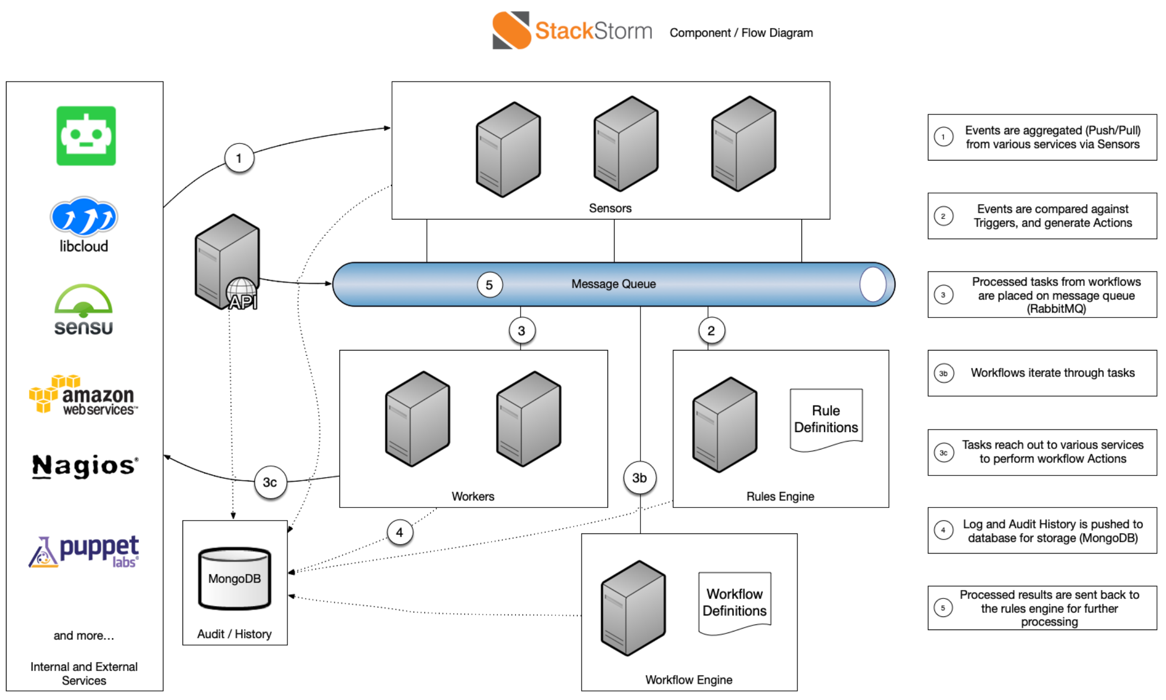
8.2.3 Conclusion

While all tools have their own advantages and disadvantages, each tool has their own use case. All tools have their own approach on automation. We have concluded that Ansible is the easiest to learn and easiest to set up. The learning curve of both Chef and Puppet is a bit steep. Both tools are harder to set up when we compare them to Ansible.

9 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.

9.1 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 Engine
Workflow Definitions
1
2
3
3b
3c
4
5
1 Events are aggregated (Push/Pull) from various services via Sensors
2 Events are compared against Triggers, and generate Actions
3 Processed tasks from workflows are placed on message queue (RabbitMQ)
3b Workflows iterate through tasks
3c Tasks reach out to various services to perform workflow Actions
4 Log and Audit History is pushed to database for storage (MongoDB)
5 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.

9.1.1 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
```

9.1.2 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.

9.1.2.1 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
```

9.1.3 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:

```yaml
---
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 %}!"
```

9.1.3.1 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.

9.1.3.2 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.

9.1.4 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 }}"
```

9.1.5 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

9.1.6 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.

9.2 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.

10 Proof of concepts

To fully test out StackStorm and check if the concept is actually useful, we're going to develop multiple Proof of Concepts (POCs) which have some sort of value for a regular organisation.

10.1 POC 1: Active Directory & JIRA

In the first proof of concept, we're going to tackle a basic yet time-consuming task and automating it via StackStorm. We're going to use JIRA - a well known and widely used ticketing/agile platform - to automatically create a user within Active Directory. The scenario would be a new employee gets hired and needs an account.

We want the following to happen:

1. A supervisor/employee/… creates an issue in JIRA - a specific type that we'll create and enters the full name of the employee.
2. StackStorm picks up on this and triggers a workflow.
3. The name entered is turned into a SAM name (Lenny Van de Winkel -> lenny.vandewinkel)
4. A random password is generated. Default passwords are bad in terms of security.
5. An account is created in Active Directory.
6. The randomly generated password gets commented.
7. The issue automatically gets closed.

10.1.1 Installing the necessary packs

First, we need to get the appropriate packages installed. A quick search of the Stackstorm Exchange reveals that we will use packages Jira and Active Directory. Installing these packs are a simple task and can be done in one of two ways:

Via the CLI

```
st2 pack install jira
st2 pack install activedirectory
```

Via the web interface
Navigate to Packs > Look up the pack > Click "Install"

10.1.2 Setting up Jira

Setting up the Jira pack is a simple but still time consuming task. The developers of the pack have a brief tutorial on their readme page.
Note**:** the tutorial still speaks of the "basic" method but this has been deprecated by Jira. The only opinion is OAuth.

We're going to assume you have basic knowledge of Jira, so we won't cover the creation of a site. If you're unsure, just visit Atlassian's website for more information. Once you have set up a website, you can begin.

We need four things to get going:

- rsa_cert_file - Path to the file with a private key
- oauth_token - OAuth token
- oauth_secret - OAuth secret
- consumer_key - Consumer key

To get these, we follow a tutorial made by Atlassian themselves. The only thing we do differently is we use a Python script in the Pack's github page to get the token and secret, rather than the Java application. Unfortunately however, the Python script is a little outdated so minor changes needed to be made:

```
#!/usr/bin/env python
# Lifted from http://bit.ly/1qpxJlj

from oauthlib.oauth1 import SIGNATURE_RSA
from requests_oauthlib import OAuth1Session


def read(file_path):
    """ Read a file and return it's contents. """
    with open(file_path) as f:
        return f.read()

# The Consumer Key created while setting up the "Incoming
Authentication" in
# JIRA for the Application Link.
CONSUMER_KEY = u'consumerkey'

# The contents of the rsa.pem file generated (the private RSA key)
RSA_KEY = read('path_to_private_key')

# The URLs for the JIRA instance
JIRA_SERVER = 'https://SITENAME.atlassian.net'
REQUEST_TOKEN_URL = JIRA_SERVER + '/plugins/servlet/oauth/request-token'
AUTHORIZE_URL = JIRA_SERVER + '/plugins/servlet/oauth/authorize'
```

```
ACCESS_TOKEN_URL = JIRA_SERVER + '/plugins/servlet/oauth/access-token'

# Step 1: Get a request token

oauth = OAuth1Session(CONSUMER_KEY, signature_type='auth_header',
                      signature_method=SIGNATURE_RSA, rsa_key=RSA_KEY)
request_token = oauth.fetch_request_token(REQUEST_TOKEN_URL)

print("STEP 1: GET REQUEST TOKEN")
print("  oauth_token={}".format(request_token['oauth_token']))
print("
oauth_token_secret={}".format(request_token['oauth_token_secret']))
print("\n")


# Step 2: Get the end-user's authorization

print("STEP2: AUTHORIZATION")
print("  Visit to the following URL to provide authorization:")
print("  {}?oauth_token={}".format(AUTHORIZE_URL,
request_token['oauth_token']))
print("\n")

while input("Press any key to continue..."):
    pass

# XXX: This is an ugly hack to get around the verfication string
# that the server needs to supply as part of authorization response.
# But we hard code it.
oauth._client.client.verifier = u'verified'
# Step 3: Get the access token

access_token = oauth.fetch_access_token(ACCESS_TOKEN_URL)

print("STEP2: GET ACCESS TOKEN")
print("  oauth_token={}".format(access_token['oauth_token']))
print("
oauth_token_secret={}".format(access_token['oauth_token_secret']))
print("\n")

# Now you can use the access tokens with the JIRA client. Hooray!
```

Follow the prompts on the script (AFTER setting up the correct settings) and copy your OAuth key and secret.

Tip: if you're having trouble with your sensor, make sure your private key file group is set to st2 and the group has permissions to read. A less-safe but easier alternative is to allow everyone to read it.

The last step is to create your configuration file with a template and fill it in.

```
touch /opt/stackstorm/configs/jira.yaml
```

Once that's done, you should have something like this:

```
---
  url: "https://goeievraagkdg.atlassian.net"
  rsa_cert_file: "/etc/jira-oauth/jira_privatekey.pem"
  auth_method: "oauth"
  oauth_token: "OAUTH_TOKEN"
  oauth_secret: "OAUTH_SECRET"
  consumer_key: "OauthKey"
  poll_interval: 30
  project: "GVK"
  verify: True
```

Now we can proceed to testing. Create an issue in your board and note down the issue key, then execute:

```
st2 action execute jira.get_issue issue_key='GVK-21'
```

Then run the command the output tells you to verify.

```
st2 execution get 622b56e96eb08c0c558d4572
```

If all went well, your output should be close to the following:

```
id: 622b56e96eb08c0c558d4572
action.ref: jira.get_issue
context.user: st2admin
parameters:
  issue_key: GVK-21
status: succeeded (1s elapsed)
start_timestamp: Fri, 11 Mar 2022 14:04:25 UTC
end_timestamp: Fri, 11 Mar 2022 14:04:26 UTC
log:
  - status: requested
    timestamp: '2022-03-11T14:04:25.265000Z'
  - status: scheduled
    timestamp: '2022-03-11T14:04:25.377000Z'
  - status: running
    timestamp: '2022-03-11T14:04:25.440000Z'
  - status: succeeded
    timestamp: '2022-03-11T14:04:26.751000Z'
result:
  exit_code: 0
  result:
    assignee: null
    created_at: 2022-03-07T14:57:45.377+0100
    description: null
    id: '10020'
    key: GVK-21
    labels: []
    reporter: Lenny Van de Winkel
    resolution: null
    resolved_at: null
    status: Being Processed
    summary: Stackstorm Testing
    updated_at: 2022-03-11T15:03:40.513+0100
    url: https://goeievraagkdg.atlassian.net/browse/GVK-21
  stderr: ''
  stdout: ''
```

Congratulations! Jira is now configured.

10.1.3 Setting up Active Directory

The Active Directory pack works by executing Powershell commands remotely. As such, the server must be configured appropriately. It's a straightforward process.

First, configure WinRM. This can be done by executing a pre-made script on the AD server

```
Invoke-WebRequest
https://github.com/ansible/ansible/blob/devel/examples/scripts/Configure
RemotingForAnsible.ps1 -OutFile "ConfigureRemotingForAnsible.ps1"
.\ConfigureRemotingForAnsible.ps1
```

Then install Remote Administration Tools by use of an action in the pack.

```
st2 run activedirectory.install_rsat_ad_powershell
hostname='remotehost.domain.com' username='Administrator' password='xxx'
```

Next, create a configuration file in /opt/stackstorm/configs named activedirectory.yaml in which we'll place our credentials. An example file can be found on the readme page. We only have one AD server, so our file looks like this:

```
---
port: 5986
transport: 'ntlm'

activedirectory:
  prod:
    username: 'Administrator@stdemo.local'
    password: 'PASSWORD'
```

That's all the configuring that's needed for now.

Once everything is configured, it's required to run "st2ctl reload --register-configs". This way all your latest configuration changes will be properly loaded. We recommend running "st2ctl reload --register-all".

10.1.4 Custom pack

To keep everything organised, we're going to put all of our elements (actions, rules, …) in our very own pack. We're going to do this very simplified by following StackStorm's tutorial. Since the steps are so simple, we're not going to cover them here. Just make sure to create everything in a folder in your own home folder (ours is /home/lenny_vandewinkel/packs/thelabpack).

For our pack.yaml, we came up with this:

```
---
# Pack reference. It can only contain lowercase letters, digits and
underscores.
# This attribute is only needed if "name" attribute contains special
characters.
ref: thelabpack
# User-friendly pack name. If this attribute contains spaces or any
other special characters, then
# the "ref" attribute must also be specified (see above).
name: The Lab Custom Pack
# User-friendly pack description.
description: A simple pack created for integration with AD & JIRA
containing a password generator and username>
# Keywords which are used when searching for packs.
keywords:
    - thelabpack
# Pack version which must follow semver format (<major>.<minor>.<patch>
e.g. 1.0.0)
version: 1.0.0
# A list of major Python versions pack is tested with and works with.
python_versions:
  - "3"
# Name of the pack author.
author: Lenny Van de Winkel
# Email of the pack author.
email: lenny.vandewinkel@student.kdg.be
# Optional list of additional contributors to the pack
```

This is self-explanatory. No explanation required.

10.1.5 Making a rule

The JIRA sensor looks for any new issue, regardless of issue type and then fires a trigger. So, we need to set up a rule so that our actions are only done when a specific issue type has been created.

In our JIRA board, we created a custom issue type called "ADDACC" with a single custom field "Full name" for the full name of the employee. So we need to create a rule that checks the issue type is "ADDACC" when the correct trigger is fired and if that's the case, execute our workflow.

In order to construct our rule, we're first going to observe the output of the trigger we want to focus our rule on (being "issues_tracker" in the Jira pack). We do this by running "st2 trigger get jira.issues_tracker". Based on this output, we can identify two outputs (payload_schema) that we want to use: issue_type (to check if the issue created is our custom issue type) and issue_name (to interact with the issue).

Start by creating a yaml file in your pack's 'rules' folder. We'll name it 'createaduser.yaml'. The contents are as follows:

```yaml
---
    name: "thelab_create_ad_user"          # required
    description: "Create an AD user with JIRA integration rule"
                                           # description is optional
    enabled: true                          # required

    trigger:                               # required
        type: "jira.issues_tracker"

    criteria:                              # optional
        trigger.issue_type:
            type: "equals"
            pattern : "ADDACC"

    action:                                # required
        ref: "thelabpack.workflow_createaduser"
        parameters:                        # optional
            issuekey: "{{ trigger.issue_name }}"
```

This means that when the "issues_tracker" trigger in the Jira pack is fired, this rule will check the "issue_type" field passed by the trigger to see if it matches "ADDACC" exactly. If it does, we'll execute the "workflow_createaduser" in our own "thelabpack" pack and pass on the "issue_name" from the trigger as a parameter to the workflow. You can name the workflow anything you want, but this is what we used.

10.1.6 Custom actions

Because we want to generate a random password for every AD user that gets created and need to convert their name to a SAM-compliant name, we need to write our own actions. So we're going to create two actions:

- One that takes an integer as input for the length of the password and then outputs a randomly generated password
- One that takes a full name as input and outputs a Windows SAM-compliant name (Lenny Van de Winkel - lenny.vandewinkel)

Creating a simple action only requires a YAML file to define the metadata and a Python or Bash file to do actual things. For our first action, we create "generate_password.yaml" and "generate_password.py" files in the "actions" folder of our custom pack and fill them as dictated in the StackStorm actions documentation.

generate_password.yaml:

```yaml
---
name: "generate_password"
runner_type: "python-script"
description: Generates a random string to be used as a password."
enabled: true
entry_point: "generate_password.py"
parameters:
    length:
        type: "integer"
        description: "Length of the password."
        required: true
```

This defines an action named "generate_password", which executes "generate_password.py" and takes "length" as a required integer parameter.

generate_password.py:

```python
import sys
import secrets

from st2common.runners.base_action import Action

class generatePasswordAction(Action):
    def run(self, length):
        return(True, secrets.token_urlsafe(length))
```

To generate a password in Python, we use Python's 'secrets' module. With the length that's passed on, we simply run the 'secrets.token_urlsafe(length)' function and directly return the output. The idea behind the use of this specific function is to just generate a random string. There's probably better ways to do this but hey - it works.

For our second action, we create another two files:

convert_name.yaml:

```yaml
---
name: "convert_name"
runner_type: "python-script"
description: "Converts a name to a SAM name for AD."
enabled: true
entry_point: "convert_name.py"
parameters:
    name:
        type: "string"
        description: "The name you want to convert."
        required: true
```

Comparable to our first action, this creates an action named "convert_name" which runs a Python script (convert_name.py) and takes "name" as required string input parameter.

convert_name.py:

```
import sys

from st2common.runners.base_action import Action

class Action_ConvertName(Action):
    def run(self, name):
        # Split the lowercase name at the first space and then join the result back together with a dot ("Lenny Van de Winkel" -> "lenny.van de winkel")
        newname = '.'.join(name.lower().split(" ", 1))
        # Then replace any spaces with an empty space ("lenny.van de winkel" -> "lenny.vandewinkel")
        newname = newname.replace(" ", "")

        return(True, newname)
```

In this script, the input name is converted to a Windows SAM compliant name. First, the name is split in two parts at the first occurrence of a whitespace and then immediately joined together again with a dot (.). Then all the whitespaces are replaced by an empty space. This result is then output.

10.1.7 Creating the workflow

Now that we can monitor for new issues, have a rule set up to filter out only relevant issues and have actions created to help us in our task, we can begin by creating the workflow.

Start by creating a "workflows" folder in the actions folder in your custom pack. This isn't absolutely necessary, but definitely encouraged to keep everything neat.

A workflow is treated as a regular action within StackStorm, so all we need is the action file itself and a metadata file (both are YAML). We'll create the metadata file "workflow_createaduser.yaml" in our actions folder and then another file with the same name in the workflows folder to be our actual workflow.

Metadata - workflow_createaduser.yaml:

```
---
name: workflow_createaduser
pack: thelabpack
description: A JIRA and AD workflow that creates a AD user automatically
and closes the ticket.
runner_type: orquesta
entry_point: workflows/workflow_createaduser.yaml
enabled: true
parameters:
  issuekey:
    required: true
    type: string
    default: GVK-X
```

This metadata file is comparable to our rule and actions. The runner type is Orquesta, which is StackStorm's own Workflow engine. The entry point is set to our other yaml file and we accept one parameter which is the issue key of the issue that triggered this workflow.

workflows/workflow_createaduser.yaml:

```
version: 1.0

description: A JIRA and AD workflow that creates a User automatically
and closes the ticket.

# A list of strings, assuming value will be provided at runtime or
# key value pairs where value is the default value when value
# is not provided at runtime.
input:
  - issuekey # We take the issue key as input, which is passed on by the
rule

# A list of key value pairs.
vars:
  - genpass: "default" # This will store our randomly generated
password, defaults to "default"
  - ename: "default" # Ename will be the name as entered exactly onto
the JIRA ticket (ename = employee name)
  - converted_name: "default" # This will be "ename" converted to a
WinSAM compliant name ("Elon Musk" -> "elon.musk")

# A dictionary of task definition. The order of execution is
# determined by inbound task transition and the condition of
# the outbound transition.
```

```
tasks:
  setup_task: # The first task is to transition the issue to "Being
Processed"
    action: jira.transition_issue_by_name issue=<% ctx().issuekey %>
transition_name="Being Processed"
    next:
      - do:
          - getcustomfields # This tells the workflow to move on to the
"getcustomfields" task
  getcustomfields: # Now we're going to retrieve the data in our custom
issue fields. In this case the employee name
    action: jira.get_issue issue_key=<% ctx().issuekey %>
include_customfields=true
    next:
      - when: <% succeeded() %> # This tells our workflow to only
proceed if the action succeeded
        publish: ename=<% result().result.customfield_10028 %> #
customfield_10028 is "Full Name", you can figure this out by running the
"get_issue" action
        do: getpass
  getpass:
    action: thelabpack.generate_password length=10 # Now we call our
custom action and generate a password
    next:
      - when: <% succeeded() %>
        publish: genpass=<% result().result %> # Publishing means
publishing output into our context. In this case, putting the output of
this action to our variable
        do: convertname
  convertname: # Here we input our full name and get our WinSAM
compliant name as output
    action: thelabpack.convert_name name=<% ctx(ename) %>
    next:
       - when: <% succeeded() %>
         publish: converted_name=<% result().result %> # Same principle
as above
         do: createuser
  createuser: # Here we finally create the user. The hostname is the
hostname of your AD server within your network (or outside of it). The
credential_name is the one you put in the config. The rest speaks for
itself. See microsoft documentation on (New-ADUser) for more info.
    action: activedirectory.new_ad_user
hostname="test-ad-instance.europe-west1-b.c.the-lab-automation.internal"
credential_name="prod" args='-Name "<% ctx(ename) %>" -SamAccountName
"<% ctx(converted_name) %>" -AccountPassword (ConvertTo-SecureString "<%
ctx(genpass) %>" -AsPlainText -Force) -Enabled $true
```

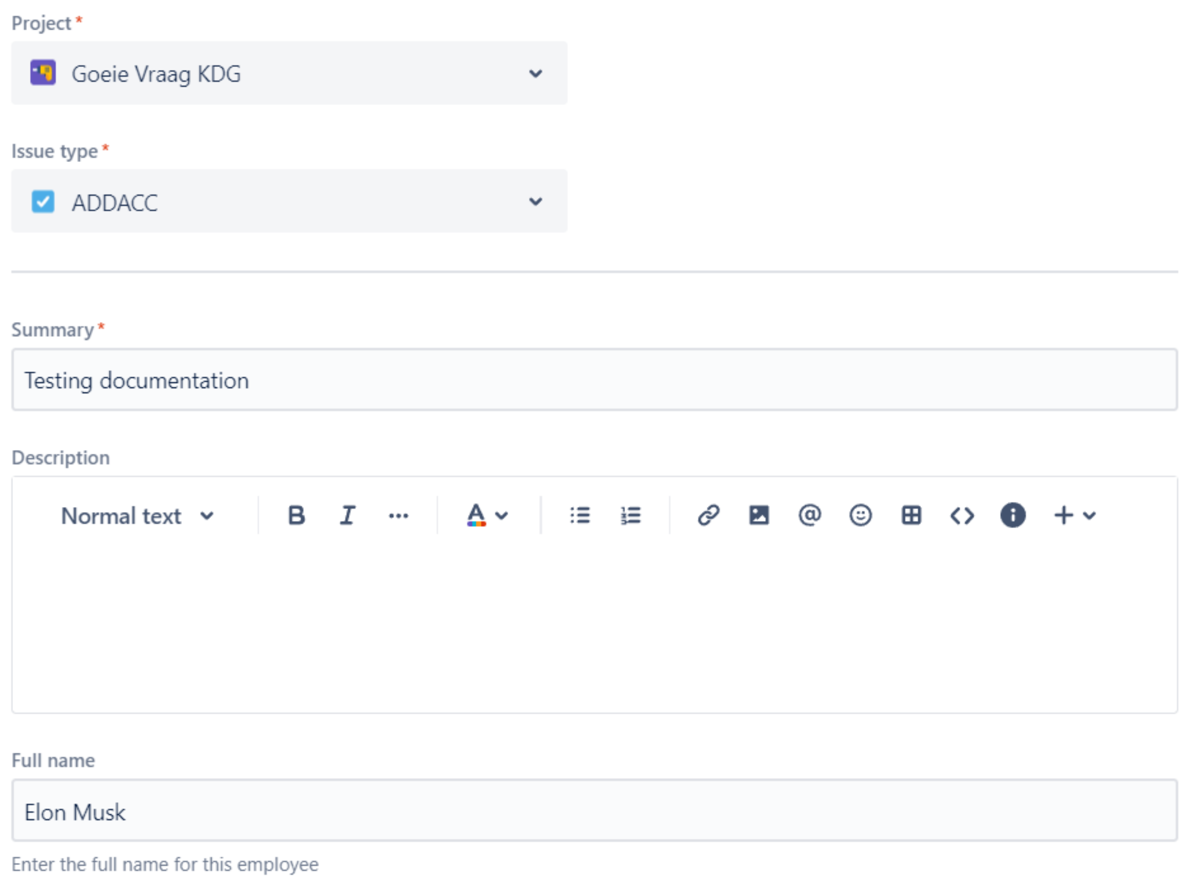
```
-ChangePasswordAtLogon $true'
    next:
      - when: <% succeeded() %>
        do: comment
  comment: # We comment the generated password
    action: jira.comment_issue issue_key=<% ctx().issuekey  %>
comment_text=<% ctx().genpass %>
    next:
      - when: <% succeeded() %>
        do: closeissue
  closeissue: # Now we set the issue to "Done"
    action: jira.transition_issue_by_name issue=<% ctx().issuekey %>
transition_name="Done"
output: # This is more for debugging than anything else. This just
outputs the values at the end of the workflow.
  #- pass: <% ctx().genpass %>
  - ename: <% ctx(ename) %>
  - convertedname: <% ctx(converted_name) %>
  - pass: <% ctx(genpass) %>
```

Comments have been added to explain every step. Info on the syntax can be found in the stackstorm documentation.

10.1.8 Finalising

After you've done all the steps, we can register the pack into StackStorm by running "st2 pack install file://PATH_TO_PACK" so in our case "*st2 pack install file:///home/lenny_vandewinkel/packs/thelabpack*". Then for good measure, reload StackStorm with "st2ctl reload --register-all".

Now go ahead and create an issue in JIRA with the correct issue type to test.



Project *
Goeie Vraag KDG

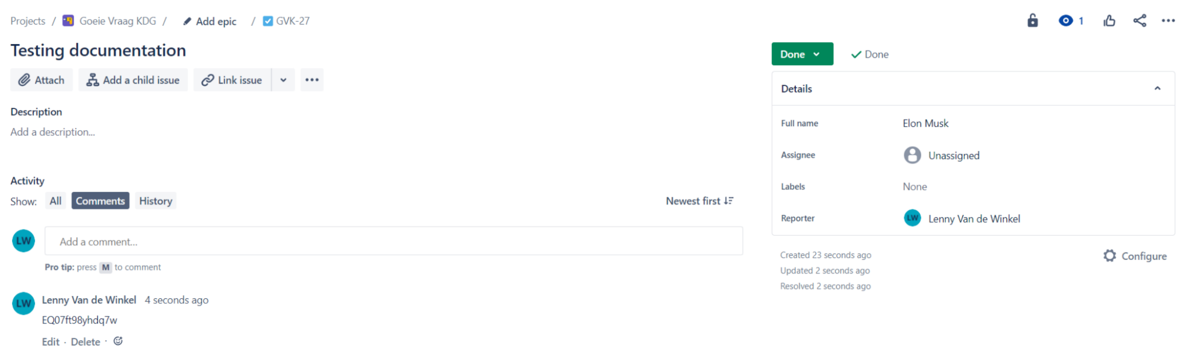
Issue type *
ADDACC

Summary *
Testing documentation

Description

Full name
Elon Musk
Enter the full name for this employee

If all is well, everything gets handled within about 30 seconds to a minute. We can confirm this by checking the issue itself, and checking the StackStorm Web UI.



Testing documentation

Description
Add a description...

Activity
Show: All Comments History Newest first
Add a comment...
Pro tip: press M to comment

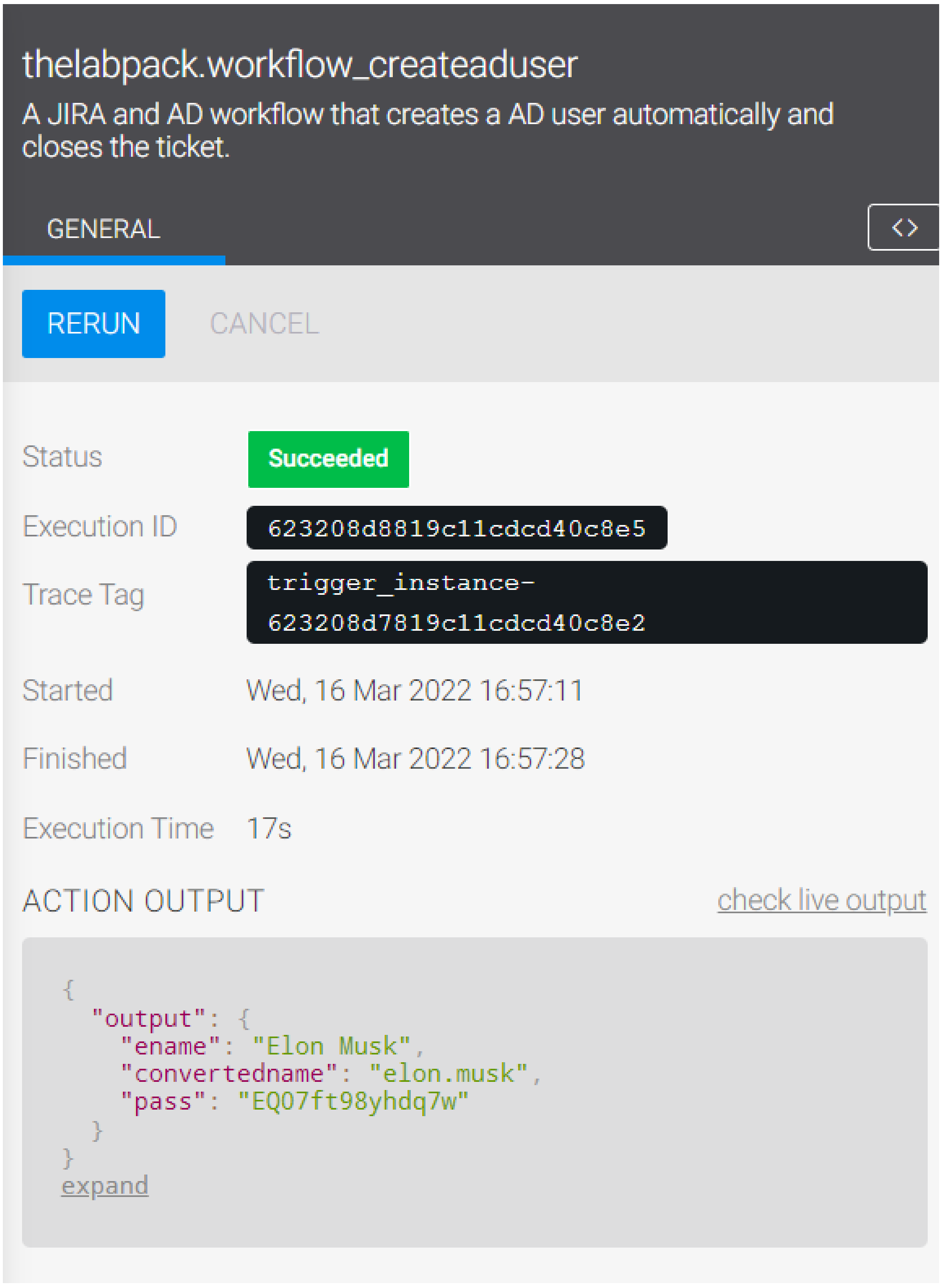
Lenny Van de Winkel 4 seconds ago
EQ07ft98yhdq7w
Edit · Delete

✓ Done

Details

Full name	Elon Musk
Assignee	Unassigned
Labels	None
Reporter	Lenny Van de Winkel

Created 23 seconds ago
Updated 2 seconds ago
Resolved 2 seconds ago


thelabpack.workflow_createaduser

A JIRA and AD workflow that creates a AD user automatically and closes the ticket.

Status: Succeeded

Execution ID: 623208d8819c11cdcd40c8e5

Trace Tag: trigger_instance-623208d7819c11cdcd40c8e2

Started: Wed, 16 Mar 2022 16:57:11

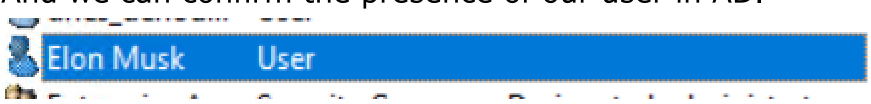
Finished: Wed, 16 Mar 2022 16:57:28

Execution Time: 17s

ACTION OUTPUT

```
{
  "output": {
    "ename": "Elon Musk",
    "convertedname": "elon.musk",
    "pass": "EQ07ft98yhdq7w"
  }
}
expand
```

And we can confirm the presence of our user in AD.


Elon Musk User

10.2 POC 2: Duo Security & JIRA/AD

In this Proof of Concept, we're going to dig deeper into security within workflows. More specifically, integrating Duo (AKA Duo Security) into a workflow.

For the scenario, we're going to create a similar workflow to POC 1. Resetting AD account passwords via JIRA. Before it gets processed, the user asking to reset the account password will have to authenticate via Duo.

10.2.1 Installing the necessary packs

Refer to POC 1 for information on how to get and configure JIRA and Active Directory. Install the "Duo" pack on top of that.

10.2.2 Configuring Duo

In order to configure Duo, follow the guide on the pack's readme.

To get the keys, go to Duo and create an application. We used "Auth API" as a template. Make sure you have users enrolled, as this POC relies on having the same users in JIRA as in Duo.

Make sure to run `st2ctl reload --register-configs` after setting up the config.

10.2.3 Workflow

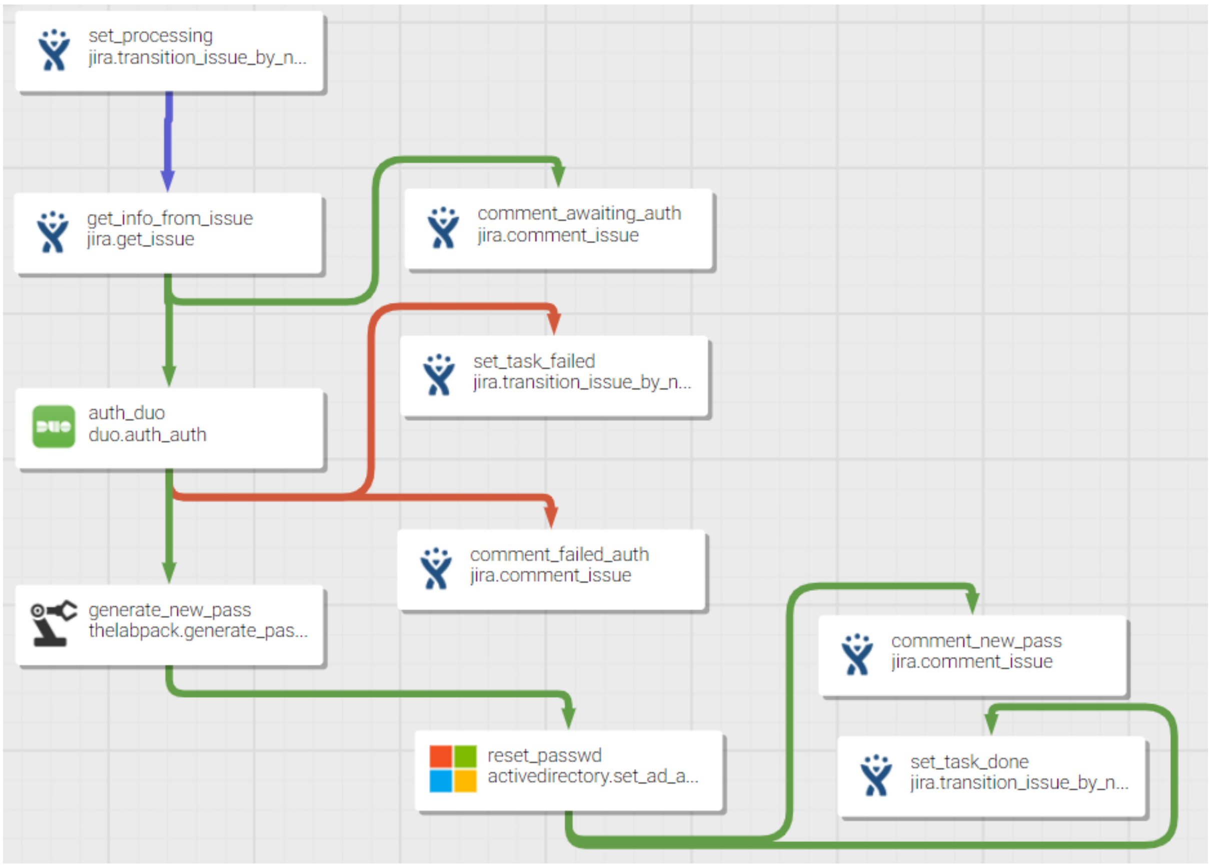
Metadata

```yaml
pack: thelabpack
enabled: true
runner_type: orquesta
name: resetadpasswd
entry_point: workflows/resetadpasswd.yaml
description: Resets a AD user password
parameters:
  issueid:
    type: string
    description: Issue key to run this on
    required: true
    immutable: false
    default: null
```

Workflow

```
version: 1.0
tasks:
  # [15, 51]
  set_processing:
    action: jira.transition_issue_by_name
    input:
      log_level: DEBUG
      issue: <% ctx(issueid) %>
      transition_name: Being Processed
    next:
      # #5b5dd0
      - do:
          - get_info_from_issue
        when: <% succeeded() %>
  # [14, 176]
  get_info_from_issue:
    action: jira.get_issue
    input:
      log_level: DEBUG
      include_customfields: true
      issue_key: <% ctx(issueid) %>
    next:
      # #629e47
      - do:
          - auth_duo
        when: <% succeeded() %>
        publish:
          - aduser: <% result().result.customfield_10031 %>
          - requester: <% result().result.reporter %>
      # #629e47
      - do:
          - comment_awaiting_auth
        when: <% succeeded() %>
  # [15, 310]
  auth_duo:
    action: duo.auth_auth
    next:
      # #d1583b
      - do:
          - comment_failed_auth
        when: <% failed() %>
      # #d1583b
      - do:
          - set_task_failed
        when: <% failed() %>
      # #629e47
      - do:
          - generate_new_pass
        when: <% succeeded() %>
    input:
      factor: auto
      log_level: DEBUG
      username: <% ctx(requester) %>
  # [277, 407]
  comment_failed_auth:
    action: jira.comment_issue
    input:
      log_level: DEBUG
      comment_text: Authentication failed.
      issue_key: <% ctx(issueid) %>
  # [279, 274]
  set_task_failed:
    action: jira.transition_issue_by_name
    input:
      log_level: DEBUG
      transition_name: Failed
      issue: <% ctx(issueid) %>
  # [282, 173]
  comment_awaiting_auth:
    action: jira.comment_issue
    input:
      log_level: DEBUG
      comment_text: Awaiting authentication.
      issue_key: <% ctx(issueid) %>
  # [15, 445]
  generate_new_pass:
    action: thelabpack.generate_password
    input:
      log_level: DEBUG
      length: 10
    next:
      # #629e47
      - do:
          - reset_passwd
        publish:
          - new_pass: <% result().result %>
        when: <% succeeded() %>
  # [289, 545]
  reset_passwd:
    action: activedirectory.set_ad_account_password
    input:
      log_level: DEBUG
      hostname:
test-ad-instance.europe-west1-b.c.the-lab-automation.internal
      credential_name: prod
      args: "-Identity <% ctx(aduser) %> -NewPassword
(ConvertTo-SecureString \"<% ctx(new_pass) %>\" -AsPlainText -Force)
-Reset"
      timeout: null
    next:
      # #629e47
      - do:
          - comment_new_pass
        when: <% succeeded() %>
      # #629e47
      - do:
          - set_task_done
        when: <% succeeded() %>
  # [566, 466]
  comment_new_pass:
    action: jira.comment_issue
    input:
      log_level: DEBUG
      comment_text: <% ctx(new_pass) %>
      issue_key: <% ctx(issueid) %>
  # [579, 549]
  set_task_done:
    action: jira.transition_issue_by_name
    input:
      log_level: DEBUG
      issue: <% ctx(issueid) %>
      transition_name: Done
input:
  - issueid
vars:
  - aduser: null
  - requester: null
  - new_pass: null
```

The transition of actions is as follows:



Blue lines are used when it doesn't matter whether the action failed or succeeded.
Red means this action executes if the previous action failed.
Green means the action executes if the previous action succeeded.

10.3 POC 3: Nagios & Twitter

In our second proof of concept we are going to make use of the Nagios and Twitter packages that are available on the stackstorm exchange website. The goal of this POC is to send a Tweet when a server goes down so that our clients know that there is a problem going on.

10.3.1 Setting up Nagios

We will first start with creating and setting up our Nagios server. We have chosen Nagios Core which is completely free. We followed the following steps to complete the installation of Nagios Core.

Once the installation is complete we need to move 2 files from our Stackstorm server to our Nagios server. These two files are the 'st2service_handler.py' and 'st2service_handler.yaml'. Both files that need to be copied are located in the /opt/stackstorm/packs/nagios/etc directory. The files need to be copied to the /usr/local/nagios/libexec directory on our Nagios server. We can leave the .py file unchanged but it's important to fill in your credentials in the st2service_handler.yaml file.

After this we'll need to configure a few extra things on our Nagios server. This is done in the /usr/local/etc/objects directory. We start by defining the command that will activate our Nagios trigger on Stackstorm inside the commands.cfg file.

```
define command{
    command_name st2nagioshost
    command_line /usr/local/nagios/libexec/st2service_handler.py
/usr/local/nagios/libexec/st2service_handler.yaml "0" "NONE"
"$HOSTSTATE$" "0" "$HOSTSTATETYPE$" "$HOSTATTEMPT$" "$HOSTNAME$"
}
```

We will then define a contact group and contact. Both will use the command that we specified in the commands.cfg file. All of this can be achieved by adding the following code to the contacts.cfg file.

```
define contact{
        name                          st2-generic-contact
        host_notification_period      24x7
        host_notification_options     d,r
        host_notification_commands    st2nagioshost
        service_notification_period   24x7
        service_notification_commands st2nagioshost
        register                      0
}

define contact{
        contact_name   stackstorm-base
        use            st2-generic-contact
        alias          Stack Storm
}

define contactgroup{
        contactgroup_name   stackstorm
        alias               StackStorm Contact Group
        members             stackstorm-base
}
```

Finally we have to define our host that we want to monitor. This can be done by creating a new .cfg file in the /usr/local/nagios/etc/objects directory. We named our file puppetclient.cfg but yours can be named differently.

```
define host {
    use                       linux-server
    host_name                 puppetclient
    alias                     puppet-client
    address                   10.132.0.36
    max_check_attempts        2
    check_interval            2
    retry_interval            1
    check_period              24x7
    contacts                  stackstorm-base
    notification_interval     0
    notification_period       24x7
    notifications_enabled     1
    contact_groups            stackstorm
}
```

As you can see from the configuration above we perform a check every 2 minutes to see that our server is still running. We have set the notification_interval to zero so that nagios will only send a notification once when the servers goes down. It is very important to always restart the nagios service when you make a change to the configuration.

The final step of our Nagios configuration is to let Nagios use our newly created .cfg file and then restart the nagios service. We do this by adding the following line inside the nagios.cfg file.

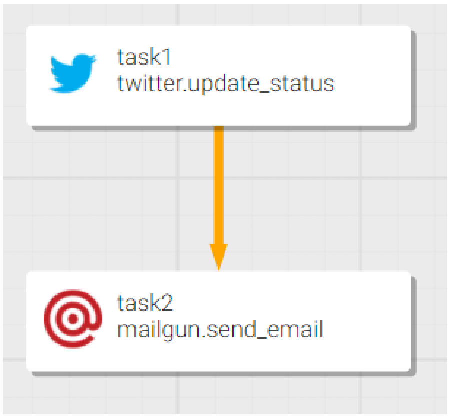
```
cfg_file=/usr/local/nagios/etc/objects/puppetclient.cfg
```

10.3.2 Stackstorm configuration

Now that our POC is set up on the Nagios server we can hop over to the Stackstorm web page. There we need to create a workflow and a rule

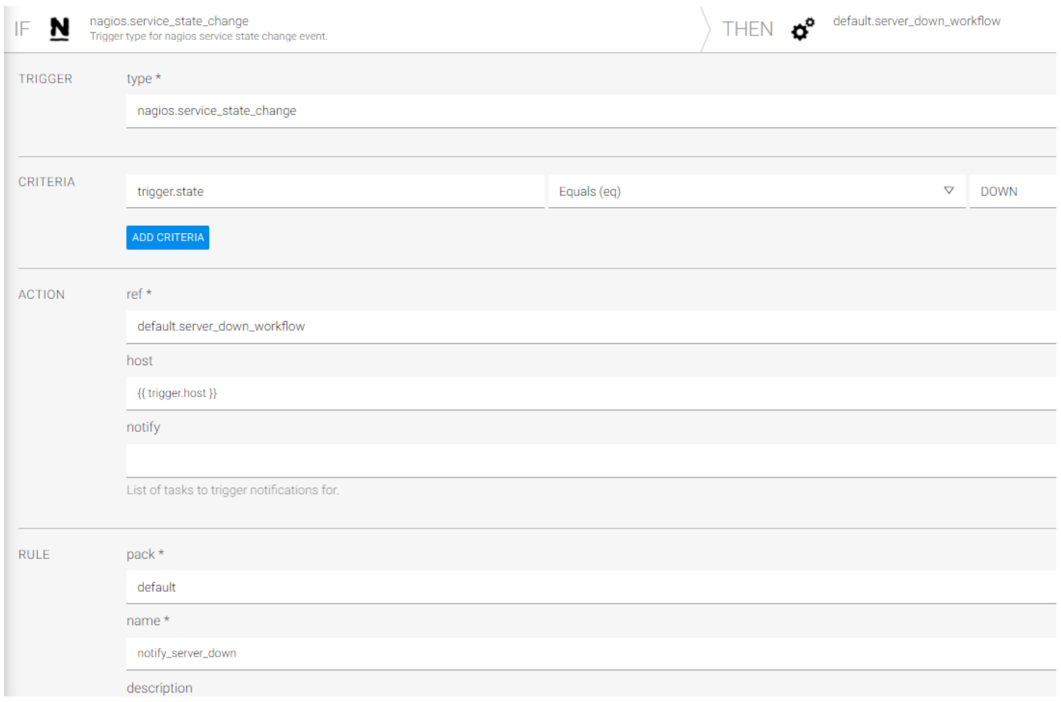
10.2.3 Creating a workflow

Our workflow for this poc will consist of 2 actions. The first action will send a tweet saying that a server has gone down. The second action will then send a mail to the admin informing them about the issue. The workflow will have one parameter containing the name of the host that has gone down. This variable can then be used in the mail so that the admin knows exactly which server is causing problems.



10.3.4 Creating our rule

We can now create a rule using the workflow we just made. We execute the workflow if the nagios.service_state_change fires and the state in the trigger payload equals DOWN.



IF nagios.service_state_change
Trigger type for nagios service state change event.
THEN default.server_down_workflow

TRIGGER
type *
nagios.service_state_change

CRITERIA
trigger.state | Equals (eq) | DOWN
ADD CRITERIA

ACTION
ref *
default.server_down_workflow
host
{{ trigger.host }}
notify

List of tasks to trigger notifications for.

RULE
pack *
default
name *
notify_server_down
description

As you can see from the image above we can give the trigger.host value to our host variable. This host variable is then used inside our workflow like stated in the workflow part of this poc.

Once we have created the rule we can save it. The rule gets automatically enabled so we don't have to restart our stackstorm service. And that's it, we now have a working POC.

10.4 POC 4: Our own Pack

What if StackStorm's sensors and actions aren't sufficient for what we want to automate? Then we can always write our own packs with sensors, actions and rules. In our third POC we will create a pack for a Linux system with a sensor, action and a rule that will do the following:

- Sensor: will check if a given volume is almost full
- Action: will extend a given logical volume
- Rule: will map the sensor with the action, so that when the logical volume is almost full, a logical volume extension will occur.

10.4.1 Pack structure

We create a folder called disk. This is the name of our pack.
The /disk folder contains the following files:

```
/actions
/sensors
/rules
/pack.yaml
/requirements.txt
/config.schema.yaml
```

10.4.2 Defining the pack

We will start by defining our pack. We define our pack in the pack.yaml file.

```
---
# This attribute is only needed if "name" attribute contains special
characters.
ref: disk
# User-friendly pack name. If this attribute contains spaces or any
other special characters, then
# the "ref" attribute must also be specified (see above).
name: disk
# User-friendly pack description.
description: Simple pack containing sensor, rule, and action.
# Keywords which are used when searching for packs.
keywords:
    - disk
    - ubuntu
# Pack version which must follow semver format (<major>.<minor>.<patch>
e.g. 1.0.0)
version: 1.0.1
# A list of major Python versions pack is tested with and works with.
python_versions:
  - "2"
  - "3"
# Specify a list of dependency packs to install.
dependencies:
  - core
# Name of the pack author.
author: Zehra Dogan
# Email of the pack author.
email: zehra.dogan@student.kdg.be
```

10.4.3 Creating the Action

An action consists of two files:

- a YAML metadata file which describes the action
- A script file with the action logic

We will call our action 'extend_lv' so we create two files named 'extend_lv.yaml' and 'extend_lv.sh' in the actions folder.

The file extend_lv.yaml looks like following:

```
---

name: extend_lv
pack: disk
runner_type: "local-shell-script"
description: Extend logical volume
enabled: true
entry_point: extend_lv.sh
parameters:
    lv_path:
        type: string
        description: Path of the logical volume
        required: true
        position: 1
    extend:
        type: integer
        description: Extend logical volume by given value.
        required: true
        position: 2
```

It's Important to choose the correct runner type. In our case, we need a local-shell-script runner because our action will be a bash script that runs on our Stackstorm server. The action will have two parameters lv_path and extend. Lv_path has position 1 which means it will be the first positional parameter $1 in the script. Extend has position 2, which means it will be $2 in our script.

Our script lv_extend.sh looks like the following:

```
#!/bin/bash

lv_path=$1
extend_by=$2

sudo lvextend -L +${extend_by}G $lv_path

sudo resize2fs $lv_path
```

This is a very simple bash script that will simply extend the logical volume located at lv_path, and then resize the file system.

10.4.4 Creating the Sensor

A sensor also consists of two files:

- a YAML metadata file which describes the sensor
- A Python script file with the sensor logic

We create two files named 'usage_sensor.yaml' and 'usage_sensor.sh' in the sensors folder.

The file usage_sensor.yaml looks as following:

```
---

class_name: "DiskSensor"
entry_point: "usage_sensor.py"
description: "Sensor for detecting high disk space usage"
trigger_types:
  -
    name: "usage_tracker"
    description: "Trigger which indicates volume is almost full."
    payload_schema:
      type: "object"
      properties:
        lv_path:
          type: "string"
        extend:
          type: "integer"
```

Here we define our trigger. We give it two parameters. Our lv_path and extend. Our Python script will inject these.

We will now talk about the usage_sensor.py script. For the full script, please visit our Git repository.

```
import eventlet
import psutil
from st2reactor.sensor.base import PollingSensor

class DiskSensor(PollingSensor):
    def __init__(self, sensor_service, config, poll_interval=30):
        super(DiskSensor, self).__init__(sensor_service=sensor_service,
config=config, poll_interval=poll_interval)
        ...


    def poll(self):
        disk_u = dict(psutil.disk_usage(self._mount_point)._asdict())
        disk_percentage = disk_u["percent"]

        if disk_percentage > self._max_percentage:
            ...
            payload = {"lv_path": self._logical_volume_path, "extend":
self._extend_amount, "count": int(count) + 1}

            self.sensor_service.dispatch(trigger="disk.usage_tracker",
payload=payload)

            ...
```

The PollinsgSensor class has a function named poll() and an attribute _poll_interval. The function poll() will be called every poll interval. In this function we will check the disk percentage. If this is bigger than a given percentage, we want to dispatch our 'usage_tracker' trigger. We also inject the payload (our lv_path and extend parameters).

10.4.5 Creating the rule

We will now create a rule in the rules folder. This rule will simply map the trigger and the action.

```
---

name: on_disk_usage_tracker
pack: disk
description: Rule firing on disk.usage_tracker when logical volume is
almost full.
enabled: true
trigger:
    type: disk.usage_tracker
action:
    ref: disk.extend_lv
    parameters:
        lv_path: "{{ trigger.lv_path }}"
        extend: "{{ trigger.extend }}"
```

Here we can also pass values to our action parameters. We want to use the values of our payload we previously set in our sensor.

10.4.6 Pack configuration

We still need to be able to pass values to our pack. For this we can make use of configuration files.

First we need to define our configuration schema in the file config.schema.yaml.

```
---

  poll_interval:
    description: "Polling interval"
    type: "integer"
    secret: false
    required: false
    default: 10
  max_disk_percentage:
    description: "Max allowed disk usage percentage"
    type: "integer"
    secret: false
    required: true
  logical_volume_path:
    description: "path of the logical volume"
    type: "string"
    required: true
  extend_amount:
    description: "extend logical volume by amount"
    type: "integer"
    required: true
  mount_point:
    description: "mount point of logical volume"
    type: "string"
    required: true
```

Our configuration consists of 5 items: poll_interval, max_disk_percentage, logical_volume_path, extend_amount and mount_point.

Once we're done with that, we can create our configuration file /opt/stackstorm/configs/disk.yaml. We fill in all the items with the values we want to use, for example:

```
poll_interval: 5
max_disk_percentage: 70
logical_volume_path: "/dev/vg00/lv1"
extend_amount: 2
mount_point: "/newstorage"
```

Do not forget to register the configuration in StackStorm's database. We do this with: sudo st2ctl reload --register-configs

Now we can use the configuration in our pack. We did so in our sensor script by using the PollingSensor's _config attribute.

```
class DiskSensor(PollingSensor):
    def __init__(self, sensor_service, config, poll_interval=30):
        super(DiskSensor, self).__init__(… config=config, …)

        self._poll_interval = self._config.get('poll_interval', None)
        self._mount_point = self._config.get('mount_point', None)
        ...
```

10.4.7 Requirements

The final thing we have to do is make the requirements.txt file. In this file, we specify which python packages we used in our scripts. In our pack, we only used the psutil package, so our requirements.txt file will just have one line:

```
psutil==5.9.0
```

The python modules specified in this file, will automatically get installed when you install the pack.

10.4.8 Finalising

Now our pack is completely ready to install and use. We install the pack with the command: st2 pack install file://path_of_pack

To test our pack we created a file on our logical volume and allocated space on it with the fallocate command. You can see the result on the demo video in our Git repository in the SelfMadePack folder.

11 Conclusion

While we first had our doubts over the capabilities of Stackstorm, we have to say that we are pleasantly surprised. It may look difficult to configure at first glance, but the Web UI makes creating rules and workflows very easy. Stackstorm is truly excellent at harnessing the power that event-driven automation offers. But of course, every piece of software will lack something. Stackstorm is mostly focused on managing already existing Infrastructure, while a tool like Ansible can also create additional resources while managing existing ones.

12 Sources

[Chef documentation](#)
[Terraform documentation](#)
[StackStorm Documentation — StackStorm 3.6.0 documentation](#)
[Jira Documentation | Atlassian Support | Atlassian Documentation](#)
[Install Nagios on Ubuntu](#)
[Stackstorm packs documentation](#)
[Nagios core documentation](#)
[Install Nagios on Ubuntu](#)
[Stackstorm & Nagios event driven automation](#)
[Stackstorm exchange](#)
[Git repository with all our code](#)