HÖGSKOLAN VÄST

2022-01-14

An open-source solution for network automation leveraged by NetBox, Ansible and RESTCONF

Alexander Birgersson
Rickard Kutsomihas

Contents

Foreword

The work distribution was equal during the system development phase, which include analysis, design, installation, coding in python and testing. In the report, the following chapters were written by Rickard Kutsomihas; 2 Vision and background, 4.3 Design and implementation and 4.4 Testing and evaluation. The chapters 3 Method, 4.1 Requirements and 4.2 Architecture and design overview were written by Alexander Birgersson.

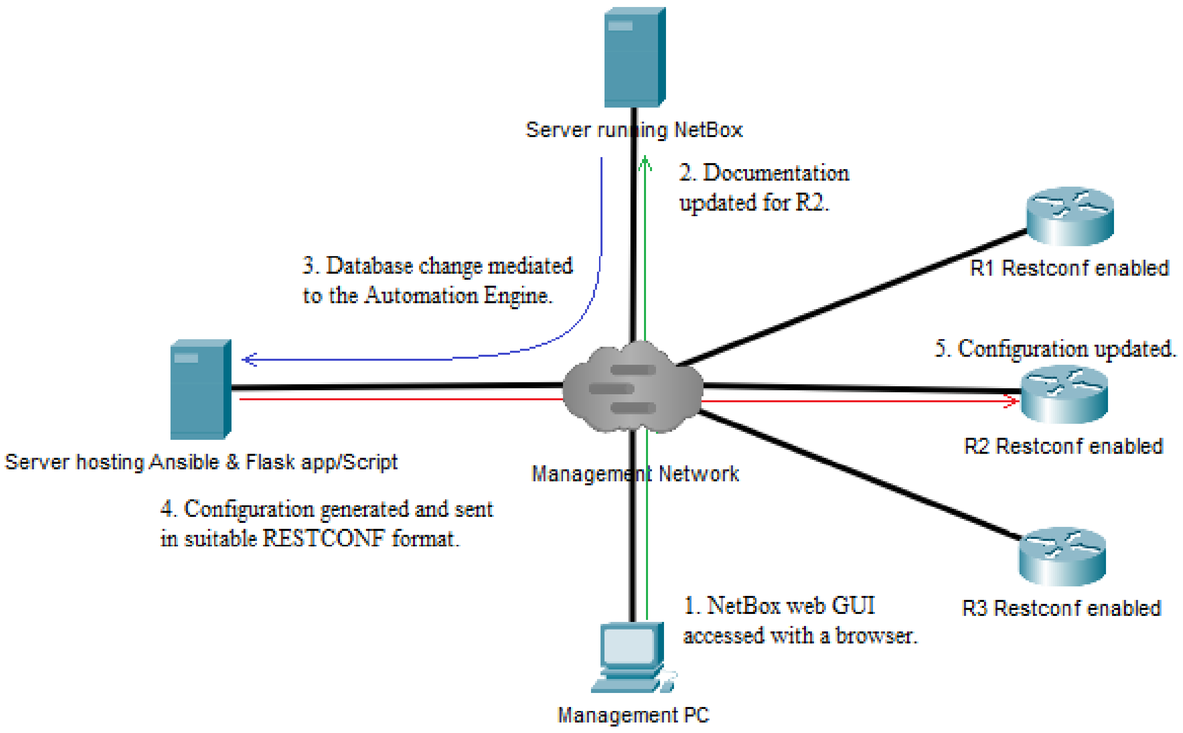
1 Introduction, 4.3.4 Implementation, 5 Discussion and 6 Conclusion were put together by both authors.

1 Introduction

The demand for centralized management and configuration of network devices in a seamless and intuitive fashion, reducing redundant configuration steps in the process, is met with several tools and solutions. Some of them are brand specific, compatible only with certain brand hardware. Having to pay premium prizes for such hardware as well as the service is not uncommon. Others are convoluted, requiring deep knowledge and some even requires the user to have advanced programming expertise, which a lot of network engineers are lacking in.

The purpose of this project is to create a centralized management and configuration solution for network equipment by using different tools and means than existing solutions, thus offering an alternative that complements their drawbacks, while also offering compatibility to a wide variety of devices. In order to achieve this the network management and documentation tool NetBox, the configuration tool Ansible and the protocol RESTCONF are utilized in conjunction with a python script.

To visualize the system, an overview is presented in Fig, 1. The management PC updates the current documentation for a RESTCONF enabled devices using the NetBox's web Graphical User Interface (GUI). The NetBox server mediates the change in its database to the server hosting Ansible and our developed python script. An appropriate configuration update will then be sent by the Ansible/script server to the RESTCONF enabled networking device. Which will reflect the changes made in the documentation.


Server running NetBox
2. Documentation updated for R2.
3. Database change mediated to the Automation Engine.
R1 Restconf enabled
5. Configuration updated.
Server hosting Ansible & Flask app/Script
Management Network
R2 Restconf enabled
4. Configuration generated and sent in suitable RESTCONF format.
1. NetBox web GUI accessed with a browser.
R3 Restconf enabled
Management PC

Fig, 1. A visual representation of the system together with a simple overview of the interaction between the nodes.

This solution is not meant to be a complete tool for management and configuration, rather a concept showcase of what can be accomplished in terms of basic configuration with free to use software and a simple python script. The idea is that the configuration is to be propagated to the network devices through managing corresponding devices in the NetBox graphical interface.

To limit the scope of this project the configuration capabilities included in this solution are as follows:

1. set or delete device hostname
2. create or delete interface
3. set or delete IP address and enable or disable interface.
4. Device configuration is set to automatically be saved to NVRAM after any change is made.

The result of this project is a solution that does offer the intended configuration capabilities using the aforementioned tools. Although some capabilities are manufacturers specific only, and the testing of the solution is limited to only one type of device.

2 Vision and background

This part of the paper consists of the chapter's 2.1 Current situation analysis, 2.2 Alternative solutions or products, 2.3 Background and theory on the subject area, and 2.4 Thesis objectif. It aims to give some context to the project and explain some of the terms used as well as provide a better understanding of the vision.

2.1 Current situation analysis

In the process of configuring network devices simple tasks typically requires several steps for the configuration part alone in addition to the verification and documentation, repeating such tasks can potentially consume a lot of time and effort. This can be remediated by streamlining the process through automating some of the steps. Network professional and former professor at Högskolan Väst Robert Andersson proposed the idea of making a python script interacting with the tools NetBox and Ansible and using the RESTCONF protocol, as these tools are already used in the field for various tasks and steps.

2.2 Alternative solutions or products

There are solutions already that does this, Cisco Meraki or available scripts and guides for the Nornir automation framework for example. However, there are some drawbacks to these alternatives; Cisco Meraki is Cisco proprietary and limited to specific hardware, which along with the subscription fees introduces an economic setback.

Using Nornir instead of Ansible is an alternative, in which case there are ready-to-use examples and guides for the intended configuration automation 0. However, Ansible uses YAML (YAML Ain't Markup Language) while Nornir uses Python. Ansible is simple by design and considered easy to understand, whereas Nornir requires understanding of Python from the user. Ansible is also the more prevalent tool at the time [2].

2.3 Background and theory on the subject area

Automation, the addition of one step in a process which will make the process more or less proceed by itself, is the translated definition Nationalencyclopedin [2] presents for the word. It also further explains that one reason for automation is to raise the efficiency and quality in a process. One of the ambitions for this thesis has been to work towards efficiency in terms of time by automating some of the steps in the configuration process [3].

In [4] DevOps is described as a culture and philosophy that "utilises cross-functional teams to build, test and release software faster and more reliably through automation". However, it seems that there is no clear and exact definition of what DevOps actually is, and some argue that definitions are not needed. It appears that collaboration between cross-funtional teams is the central part of this organizational approach [5].

Single source of truth is described by Wikipedia.org as "In information systems design and theory, single source of truth (SSOT) is the practice of structuring information models and associated data schema such that every data element is mastered (or edited) in only one place."

NetBox is described as "an infrastructure resource modelling (IRM) application designed to empower network automation." in [6]. It lets the user construct digital models of network infrastructures and network devices in a physical- and logical topology documentation manner. It is intended replicate the real world to and to serve as a source of truth, representing the desired state of a network rather than its operational state. NetBox also has a webhook feature, which is a mechanism that triggers on certain specified events in NetBox and sends the event information to a specified webhook receiver as a HTTP request with a GET, POST, PUT, PATCH or DELETE method.

Ansible is an open-source community project automation platform sponsored by Red Hat and according to [7] it is "the only automation language that can be used across entire IT teams from systems and network administrators to developers and managers". Ansible is designed to be a simple IT automation engine which uses its own simple language YAML (YAML Ain't Markup Language), intended to be structured in such a way that it approaches plain English while describing automation jobs. Ansible uses a module architecture where different modules can be applied for different purposes, command modules, inventory modules or network modules for instance, which is meant to simplify setting up tasks by making it more specific through these modules.

RESTCONF is a HTTP-based protocol that provides a programmatic interface that can access data defined in YANG. YANG is described in RFC7950 as "YANG is a data modeling language used to model configuration data, state data, Remote Procedure Calls, and notifications for network management protocols." It was originally designed to model configuration and state data manipulated by NETCONF (Network Configuration Protocol) but from YANG version 1.1 it was extended to also include use for the RESTCONF protocol. This version also proposed extending the encoding options from only XML (Extensible Markup Language) to also include JSON (Javascript Object Notation), which are different data exchange formats where data is structured and written in accordance with the format. The formats are designed for interoperability which ensures that data can be exchanged between different applications and be structured the same way regardless [8]. RESTCONF leverages the data store concept defined in RFC6241 for the NETCONF protocol, where these configuration datastores can be accessed through CRUD (Create, Read, Update, Delete) operations. With these operations network devices state data and configuration data can be retrieved, and the configuration data can also be modified.

A digital twin is a concept where a physical object is connected to a digital object in such a way that there is a data flow between the objects in both directions. If something is changed in one of the objects, the change should be reflected in the twin object. For example, in one scenario if one of the objects lights up an indicator light the other object should also light up a corresponding indicator light. The digital twin does not necessarily have to be a 3D replica of the physical object. A digital shadow is a similar concept to the digital twin concept. Here the dataflow is unidirectional, and only flows from the physical object to the digital object, in other words to the digital shadow [9].

2.4 Thesis objectif

The vision of this project is to streamline and simplify the configuration process of network devices. The general idea is to use NetBox as a single source of truth, pushing relevant configuration to network devices in accordance with the resource models in NetBox. By using the NetBox webhook function that sends information about changes made to the models to a server the information can be used to build configuration information and deliver it to the devices. Tedious and time-consuming configuration steps performed in manual configuration will be performed automatically in the background while interacting with the NetBox GUI.

By approaching the problems with a DevOps philosophy in mind the plan is to engage with NetBox, Ansible and RESTCONF and evaluate which of their operational possibilities can be translated to relevant configuration data in relation to these tools and the network devices. Then to develop the script needed for conveying this data from NetBox to the intended devices.

For the interactions between the physical objects and the digital objects the vision is twofold. This work will focus on having the NetBox resources act as digital objects and the network devices as physical shadows, basically inverting the digital shadow concept. The extended vision of this concept is to further the work on this project and transition from an inverted digital shadow concept to a digital twin concept where the network devices are the physical twins and the NetBox resources are the digital twins.

3 Method

This section describes the processes that took place before the construction of the solution. This is an important step as it derives what the end-product will look like. The steps conducted in the planning process are presented below, with the contents of; 3.1 how the requirements of the system were established, 3.2 analysis of the requirements, 3.3 software design and implementation process, 3.4 development and deployment decisions, 3.5 how the testing and evaluation are to be performed.

3.1 Collecting requirements

The requirements of the system were provided by Robert Andersson who, initially, has proposed the solution. This was mediated by Thomas Lundquist, teacher at University West. Further contact with Robert about additional requirements or details about the solution was not conducted, as the proposal was deemed to be more general in nature.

3.2 Analysis

In order to build a system that satisfies the intentions of network automation, it is important to gain a good understanding about the programs and protocols at play and identify their interfacing capabilities.

Because no existing system was presented, the analysis process was based solely on the software requirements provided. Consisting of an information gathering process to gain an understanding of the “must have” components and their dependencies. This revolves around searching the web, reading documentation, looking at videos, and performing labs on Cisco Devnet.

3.3 Design and implementation

The system-to-build is depicted using a dataflow diagram as a visualization of the interactions between components. The script itself and its interaction with the other components is designed by creating a flowchart based on the wanted functionalities, so that an understanding of the intended workflow can be established. Multiple flowcharts were constructed as the feasibility in terms of design was a bit uncertain. These flowcharts can be viewed in Appendix A: Flowcharts. The flowcharts cover different design options; one provides a more user-friendly experience with the intent of an interface that prompts the user to enter user-specific data to be used in the script (potentially using a NetBox plugin) without having to access the source code. Another flowchart explored the use of a configuration file and a log function.

Another consideration was how to deal with potential inconsistency between the information existing in NetBox and the actual configuration running on a device. For example, what if a user creates a new interface for a device in NetBox but due to a network or device problem, the configuration sent never gets executed on the unit. Then the device documentation in NetBox does not correlate with the configuration running, and an inconsistency between the two has emerged. Since Netbox is a tool that is meant to be used as source of truth, this is obviously a problem and would need to be address in some form. In such a scenario, our intention was to have the script revert the change made in NetBox.

Different deployment and corelated designs options were also explored, such as whether the solution should be designed to be more tailored to be used in a dedicated management network or for in-band network management. For example, in an in-band deployment, multiple IP-addresses could potentially be used for establishing communication to one of the devices' Layer 3 interfaces. In that case, the script could try to connect using another IP-address if it failed with the first. While in an out-of-band deployment typically one interface on the device is used as a dedicated management port, this would not be needed.

A decision was made to focus only on implementing basic functionalities to the script at first, due to time constrains mainly. Basic functionalities are considered to be user supplied input in NetBox, that can be directly correlated to a configuration executing on a router or switch. This includes configuration of hostname, interface and IP-address. Extended functionality will need to be implemented in future development or by the user to suit their needs, as the script will be released as open-source.

Ultimately, a flowchart that depicted a fundamental and basic approach without extra functionalities was chosen as the design goal. After the completion of the flowchart, a pseudocode was produced to gain a more detailed idea about the how the code might be constructed.

3.4 Development and deployment

Through our continuous discussions, a different development environment was considered. The choices involved the following:

A) Using a local computer at University West and have it connected to physical routers and switches in a lab environment.

B) Using a virtual machine running on the University Wests' VMware cluster and perform test configuration against one of the always-on sandbox routers provided by Devnet.

C) Using a VM to simulate a lab environment running GNS3 or similar program.

Choice B was selected as it provided an environment that could be easily accessed by both developers and didn't include occupying physical lab equipment or running a simulation (which also required obtaining a supported OS for the networking devices).

Because this management solution wasn't requested to be used in an existing environment, it will not be deployed in a production network as a part of this work.

3.5 Testing and evaluation

The testing and evaluation of the scripts' implemented functions is occurring as a part of the development processes, as new functionality is introduced. When the development is completed, and if time allows, further testing and evaluation will be conducted on campus at University West computer lab, where the script will run against physical routers and switches to determine the interoperability in a more "real world" scenario.

A more dedicated testing process using external testers was not explored because of the specific use case of the implementation. Another reason was that the script functionalities were deemed to be sufficiently testable without using a more dedicated process.

4 System development

This chapter is related to the development of the system. It correlates with the chapters presented in 3 Method. It contains the 4.1 Requirements, 4.2 Architecture and design overview, 4.3 Design and implementation and 4.4 Testing and evaluation.

4.1 Requirements

The system-to-build should fulfill the following requirements. Network automation should be achieved by the development of a software/script, which will bring together the functionalities of the software components. The software components consist of NetBox, Ansible and RESTCONF.

When a user interacts with a device in NetBox, the configuration running on that device should be updated to reflect the changes made in the documentation. Ansible are to be used as the configuration tool and the configuration should be sent to the networking device with RESTCONF.

The end solution should therefore meet the requirements stated below:

- Develop a software/script to interconnect the functionality of the software components.
    - NetBox
    - Ansible
    - RESTCONF

- A user should be able to automatically configure networking device from the NetBox GUI, without added complexity.

4.2 Architecture and design overview

NetBox and Ansible are only compatible with Linux as the underlaying Operative System (OS). It was clear that a Linux system was needed. NetBox is tested to work on Ubuntu 20.04 and CentOS 8.3 at time of writing. Ansible is compatible with many different Linux distributions. The Linux server can host both applications if it's running CentOS 8.3 or Ubuntu 20.04. Ubuntu Server 20.04 was chosen as the OS to be used over CentOS 8 as it will reach its end-of-life in December 2021 [10]. Using an Ubuntu system therefore felt more relevant, as we believed this will be the more prominent choice between the two in the future.

As for the networking devices that will be automatically configured, they must support configuration via RESTCONF. The development environment uses a Cisco csr1000v router for programmatic testing, which is publicly available on the Cisco's Devnet platform. The router is running IOS-XE 16.9.3 as its OS.

In order for NetBox to advertise when changes are made to devices in its database, a build-in function called webhooks will be used. Webhooks mediates the documentation change by sending a HTTP request to a webserver. When a NetBox user interacts with the documentation of a device through the GUI, and either creates, updates or deletes information, it will trigger a webhook to be sent.

Different models exist in NetBox and each model stores its own type of objects in the database. Webhooks was enabled for models that are we perceived to be configuration related, such as the device, interface and IP-address model. The Webhooks will inform another software/system when changes are made to a relevant object. Such as documentation changes for a device's hostname, interface or IP-addresses. The webhook is sent by HTTP and has a payload in JSON format, which contains information about the object, such as its data before the change occurred and the data after the change.

To interconnect the software components, a Python script was developed. Python was a suitable choice as it's a programming language that doesn't need to be compiled. Therefore, scripts can easily be created and tested. Both developers had previous experience with the programming language as well.

The script utilizes the Python libraries Flask and Requests. Flask is a framework for web application development [11]. Flask provides a Web Server Gateway Interface (WSGI), which allows received HTTP data to be presented into the script. When running the script as a Flask application, a small-scale webserver build-in to Flask listens for incoming HTTP traffic destined for a specified URL. When the data is presented into the script it triggers a function to run in the Python code. This will allow our script to execute when it receives a webhook from NetBox. The Requests library is used to allow the script to initiate its own

HTTP requests, which will be needed to retrieve the IP-address of a device from the NetBox API if it is not included in the webhook.

Ansible uses the concept of playbooks to execute commands on remote devices. A playbook is a YAML file with user specified instructions and commands. The instructions tell Ansible which hosts from the inventory file the playbook should be executed against, along with which Ansible plugin should interpret the commands and be in charge of the communication. Ansible comes with a set of plugins as default, and additional plugins can be downloaded from Ansible Galaxy. The netcommon plugin collection contains a plugin that allows Ansible to support the use of RESTCONF in a playbook, namly the "netcommon.restconf_config" plugin [12].

Playbooks are a user-friendly way to present configurations to the Ansible software, and allows them to be acted upon. However, managing playbooks files felt cumbersome when included in an automated process. Because Ansible is written in Python, it can be interfaced by its Python API, as described in [13]. This allows our Python script to interface with the Ansible source code and to run our playbooks as Python code instead, which circumvents the use of playbook files. Ansible and the script need to exist on the same machine because of this. NetBox can exist on an external server as long as HTTP traffic can flow to/from the server running Ansible/script.

In short, the solution consists of:

- NetBox (can reside on the same server as Ansible, Flask and the python script)
  - Webhooks to communicate when changes are made.
- Python script
  - Executes as a Flask application to run a python function when receiving a webhook.
  - Processes the data from the webhook and interfaces with Ansible to run it as a play.
  - Uses Requests to query NetBox for the IP-address of a device, if needed.
- Ansible
  - Uses the netcommon.restconf_config plugin to send configuration to remote devices as RESTCONF.

To gain a better understanding of how the system interacts between its external components, a topology is presented in Fig, 2. The steps below explains the communication.

1) The management PC uses a browser to access the NetBox GUI and updates the documentation for one of the RESTCONF enabled devices, in this case R2. By either changing the hostname, create/update/delete an interface or assign/remove an IPv4 or IPv6 address to one of the interfaces.

2) NetBox will inform the Ansible/script server about a change to one of its objects, by sending a webhook. The webhook is a HTTP POST request and contains data about the object in JSON.

3) If the webhook doesn't include a management IP-address to the target device, R2 in this case, it will retrieve it by issuing a HTTP GET request to the NetBox API.

4) The Ansible/script produces an appropriate configuration based on the webhook data and sends it in a RESTCONF suitable format to R2. The payload is JSON data in a structure that complies with the target YANG data model. It is send as a HTTP POST, PATCH or DELETE request method depending on if a setting needs to be created, altered or removed on R2.

5) R2 receives the configuration and updates its running configuration via its RESTCONF programmatic interface, as informed by the received payload. The configuration active on R2 now reflects the changes that was made to its documentation in NetBox.

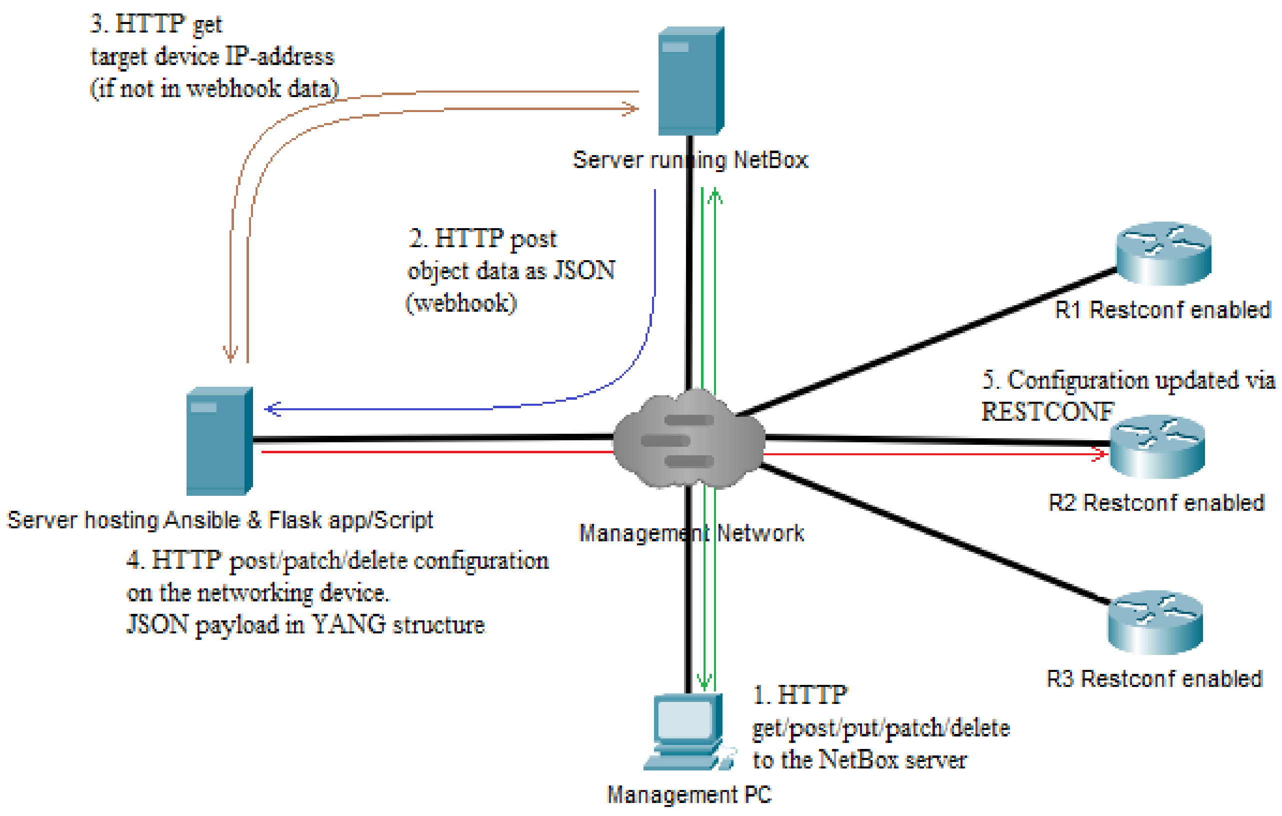


Fig. 2. The management PC accesses the NetBox GUI via its browser and makes a change in the documentation for R2. NetBox will inform the Ansible/script server about the change by sending a webhook. The Ansible/script server retrieves the management IP-address to the device, if needed, and sends a configuration update to R2 based on the webhook data.

4.3 Design and implementation

This chapter is dedicated to describing the finished product and how it was created with mentions and explanations of some details. It also includes design choices and the reasoning behind them as well as the implementation of the idea behind this paper. The chapter is split up into 4 subsections: 4.3.1 The finished product, 4.3.2 The creation processes, 4.3.3 Design choices and 4.3.4 Implementation.

4.3.1 The finished product

The finished product is the entire solution of using NetBox, Ansible, Flask and RESTCONF together with the python script made for this paper in order to configure network devices. In this case finished means that the solution as a whole works to a certain extent. But perhaps it can be improved upon by further work on the script or by exploring the potential of utilizing the other tools mechanisms to extend the solution functionality.

To use the product the user should first read the manual and make sure all prerequisites are met. If they are then it's only a matter of navigating the NetBox UI and manage network devices. If changes made in NetBox can be translated into actual configuration on the devices, changing the IP address on an interface for example, NetBox will send a webhook containing information to a server. The webhook will be received by the Flask app where the script is located, and the script will process the webhook and either accept it or ask NetBox for more information via specific API calls. The information will then be translated to device configuration by the script into a format compatible with Ansible and RESTCONF, and then push the configuration with the Ansible Python API to target device. A message will be sent back to the server detailing whether the configuration attempt was successful, unsuccessful or if the target device could not be reached.

The Python script that was created could also be described as a product since it was produced specifically for this thesis by the writers. The script simply gathers data and decides whether it can be translated into device configuration or not by comparing the data to predefined conditions. If so, it then translates and formats the data into configuration information that the network devices can understand and execute. The script also assists in sending the data to the intended devices, as well as provide the results of the configuration attempt.

4.3.2 The creation processes

Creation of the product started with gathering information across the internet about the tools used in the project, such as manuals, guides, and requirements. Then flowcharts were drawn to create a visual representation of the processes and components in the solution and how the data flows between these. Several flowcharts were drawn with consideration to different approaches and functionality, with Fig. 3, representing core functionality and logging, which established a baseline to start from. The resulting product operates in accordance with Fig. 4, which excludes a logging mechanism and uses the Ansible Python API to forward device configuration rather than creating and/or running Ansible Playbooks. Alternative flowcharts with different ideas and functionality can be viewed in Appendix A: Flowcharts.

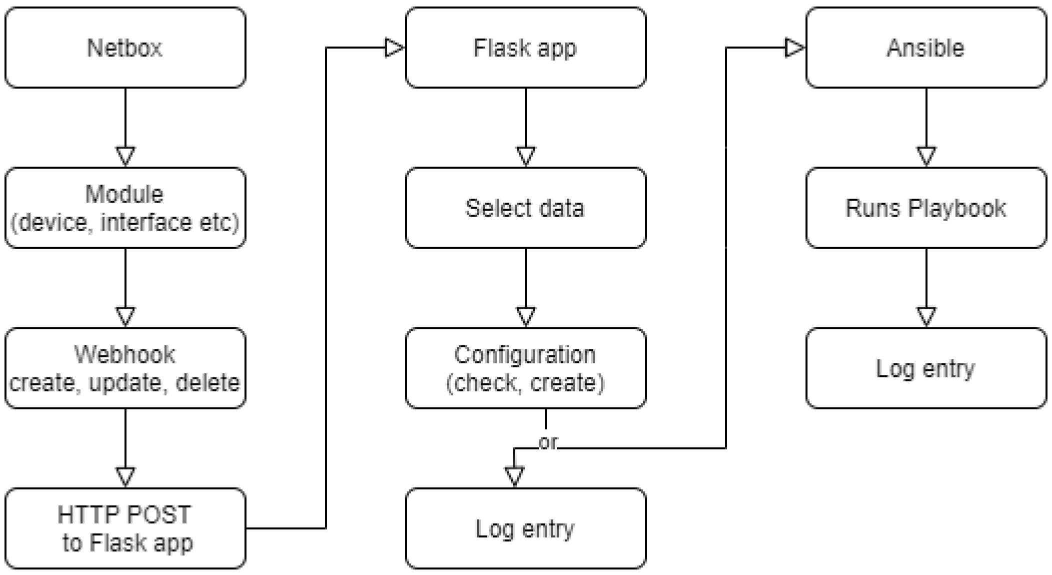

Netbox
Module (device, interface etc)
Webhook create, update, delete
HTTP POST to Flask app
Flask app
Select data
Configuration (check, create)
or
Log entry
Ansible
Runs Playbook
Log entry

Fig. 3. The flowchart that was used as the design goal.

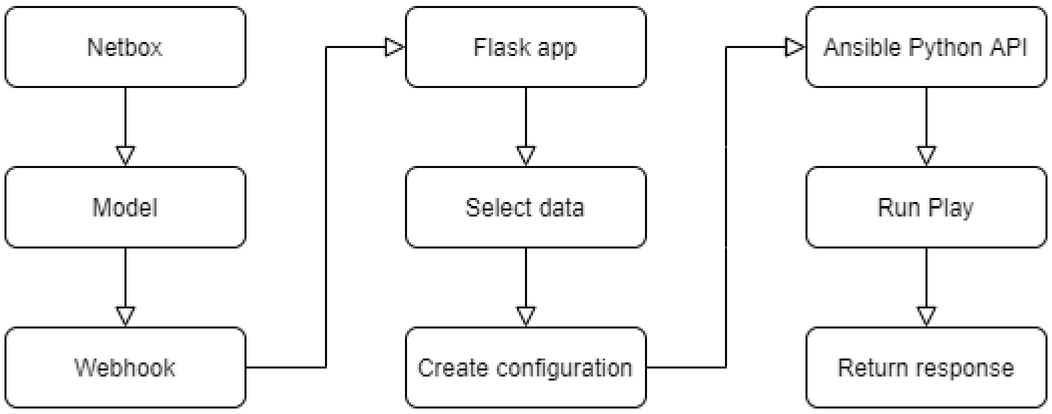


Fig. 4. The actual workflow of the developed solution. Uses Ansible python API instead of playbook files. The separate logging functionality was omitted.

After drawing flowcharts pseudo code was written to accommodate the flowcharts and to aid in structuring the actual code. The pseudo code can be viewed in Appendix B: Pseudo-code. The flow charts and pseudo code in Appendix A and Appendix B were designed with the goal of this project in mind but also extended with ideas that might improve the concept with added functionality and features, in case of having enough time or easy implementation or as a reference for future work.

Gaining knowledge and experience on the tools used in the solution was an important step of the product creation process, predominantly the applications NetBox and Ansible, where the focus was to get insights to how these programs operate and what mechanisms and features could be leveraged. To better understand how Ansible works learning labs were conducted with the application on Cisco Devnet at developer.cisco.com. To gain a better understanding of NetBox a hands-on experience approach was used to get acquainted with the application and gauge what resources could be benefited from, and what mechanisms could be considered when translating actions into device configuration, with an intuitive aspect in mind. Since the source of the configuration would be NetBox limitations were established with that in mind by engaging with the UI, the webhook function and the NetBox API.

The script, which can be viewed in Appendix C: Python script, was created in one file together with the Flask app. The programming started first after engaging with the tools and limitations had been established. The code can be divided into three main segments: the variable segment, the function segment, and the execution segment. The variable segment consists of user definable settings such as paths to applications and files. It also consists of the predefined configuration limitations in a dictionary/list format. This segment is designed with the intention of being easy to understand and edit so that users can set their own settings and increase or decrease configuration limitations if they wish to do so. The execution segment begins with the Flask app function of listening to HTTP requests, in this case webhooks, and processes these in a step-by-step manner to end up either translating them into device configuration or discarding them. Functions in the function segment are referred to in these steps. The approach for these functions is to search for relevant information in the webhooks by comparing the prevalence of terms corresponding to identical terms in the configuration limitations list. Then to deconstruct the webhooks into a more manageable dictionary format with only necessary information. Then to add information if necessary. And finally to utilize the information by constructing it in a format that is suitable for device configuration.

4.3.3 Design choices

The product is not dependent on Flask in order to work, any other WSGI could possibly suffice. The choice of using Flask is simply due to it being described as simple and minimalistic by design and easy to use and implement. Since the only function needed here was a listener for http requests in the form of NetBox webhooks anything more advanced was considered unnecessary.

The choice to exclude the logging function from the product was due to the decision to focus on core functionality since time constraints and low programming experience was considered a possible issue, so a prioritization order was established. The intended original function of creating and/or running Ansible playbook files to configure devices was replaced in favour of using an Ansible Python API instead. The reason for this is that when comparing the options, omitting the creation and usage of files with no apparent impact on function seemed like the better option.

Design of the actual script in how it processes the data is probably not very optimized. Instead of stripping the webhooks of relevant data and putting it into new contexts and constructing new lists and dictionaries to build configuration information from, the webhooks original structure could most likely have been kept as is and search mechanisms used to gather the relevant data. The choice to deconstruct the webhooks into simpler forms was made during the script creation process in order to make it easier to understand and handle the data.

4.3.4 Implementation

One difficult part of implementation was to use borrow the Ansible Python API example code and first of all make sense of it and consider what parts of it are useful or not for this application, but most of all change it in accordance with this application. A lengthy trial and error process ensued trying to figure out what works and what doesn't, specifically when interacting with the CSR1000v DevNet router. In addition to this the trial and error process was further extended when trying to structure the RESTCONF Ansible play format so that the intended configuration would be executed on the device. Exploring what resource paths on the device could be leveraged was a time consuming and information on the internet was scarce and unsatisfiable. Additionally what RESTCONF methods to utilize was also not apparent, and information from guides did often not coincide with findings from the trial and error process. For example one guide leveraged the PUT method in one scenario, which was unsuccessful when used in our case, but successful when replaced with a POST. All of these inconsistencies and the multiple areas with uncertain format made the whole process quite difficult and time consuming.

Also, the Ansible plugin for RESTCONF didn't seem to be compatible with the configuration saving module of the Cisco-ai YANG model. As it didn't activate the RPC on the networking device and thus not activating the instruction on the device. To solve this, the script sends an independent HTTP POST request to save the configuration, targeting that YANG module, after having ran Ansible. Because Ansible was initialized and has already loaded the device credentials from its variable file (Var file) it can still be called upon to allow a reuse of the authentication credentials to be submitted in the HTTP header.

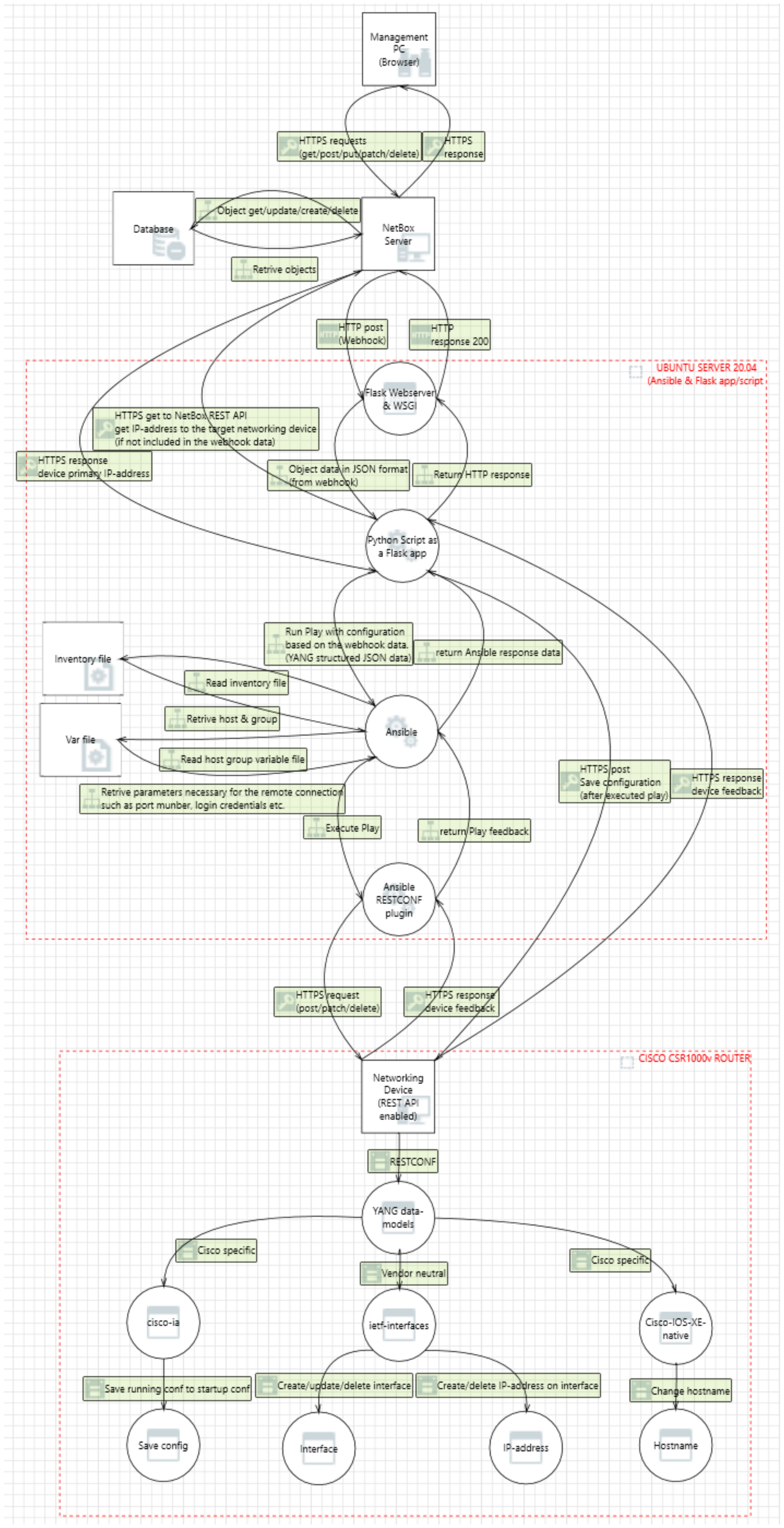
The workflow of the system can be described by the data-flow diagram presented in Fig, 5. The red squares indicated files and software components that need to exist within the local device. The management PC, NetBox server and database can reside on other devices.

From the top:

1) The management PC interacts with the NetBox web application. Which in turn interacts with its database and sends a webhook when there is a new change to an object belonging to either the device, interface or IP-address module.
2) The webhook is received by the Flask built-in webserver and its JSON data is presented to the script via the WSGI.
3) If the management IP-address (primary IP-address) for the target device isn't included in the webhook it will be retrieved from the NetBox API. An Ansible play is constructed based on the data from the webhook.
4) Ansible is called upon to run the play through its python API. Ansible loads the inventory file to determine if there is a valid host entry that matches the management

IP-address. If so, determines which group it is a member of. Ansible loads the group variable file which consist of connection parameters and credentials for all the devices belonging to that group.

5) The Ansible plugin interprets the play and is responsible for initializing the communication to the target device.
6) The RESTCONF enabled networking device receives the configuration payload. A RESTCONF device has multiple YANG data-models that each consist of different configurable values. The YANG data-models that are interacted with through Ansible are ietf-interfaces and Cisco-IOS-XE-native. The device performs the instruction and responds back with feedback from the operation.
7) Ansible reports back feedback from the operation.
8) An independent HTTP POST request is sent to the networking device to save the configuration to NVRAM, targeting the Cisco-ai YANG data-model. The device performs the instruction and responds back with feedback from the operation.


Management PC (Browser)
HTTPS requests (get/post/put/patch/delete)
HTTPS response
Database
Object get/update/create/delete
NetBox Server
Retrive objects
HTTP post (Webhook)
HTTP response 200
UBUNTU SERVER 20.04 (Ansible & Flask app/script)
Flask Webserver & WSGI
HTTPS get to NetBox REST API get IP-address to the target networking device (if not included in the webhook data)
HTTPS response device primary IP-address
Object data in JSON format (from webhook)
Return HTTP response
Python Script as a Flask app
Run Play with configuration based on the webhook data. (YANG structured JSON data)
return Ansible response data
Inventory file
Read inventory file
Retrive host & group
Var file
Ansible
Read host group variable file
Retrive parameters necessary for the remote connection such as port munber, login credentials etc.
Execute Play
return Play feedback
HTTPS post Save configuration (after executed play)
HTTPS response device feedback
Ansible RESTCONF plugin
HTTPS request (post/patch/delete)
HTTPS response device feedback
CISCO CSR1000v ROUTER
Networking Device (REST API enabled)
RESTCONF
YANG data-models
Cisco specific
Vendor neutral
Cisco specific
cisco-ia
ietf-interfaces
Cisco-IOS-XE-native
Save running conf to startup conf
Create/update/delete interface
Create/delete IP-address on interface
Change hostname
Save config
Interface
IP-address
Hostname

Fig, 5. Data-flow diagram over the system's workflow.

4.4 Testing and evaluation

All the system requirements have been met completely or to some degree. Some of these requirements were to use specific software, namely NetBox, Ansible and RESTCONF. As can be seen in Table. 1, these requirements have been met completely. To reach network automation has been marked as a partly met requirement since full automation has not been reached. To create a script that works as intended is also a partly reached requirement. This is because although the script utilizes the tools as intended and produces the results that were intended, it only works completely with certain devices since the test device used as a reference lacked the sufficient RESTCONF resources to create and test a more open standard approach.

Table. 1. Displays the degree of requirement fulfillment.

Requirements	Not met	Met partly	Met
Use NetBox			✓
Use Ansible			✓
Use RESTCONF			✓
Reach Network automation		✓	
Create script that works as intended		✓	

The main problem posed in this project has been solved. A solution for centralized management and automated configuration has been established. Whether it has actually improved the situation for anyone is not yet answered, but even if one would find the solution lacking it still serves as an example that free and open-source software can be leveraged to successfully create a centralized network device management that automates configuration tasks.

5 Discussion

Robert Andersson requested a solution for automatic network configuration using the software components NetBox, Ansible and RESTCONF. When a user alters the documentation in the NetBox GUI for a network device, such as a router, the configuration running on the device will be automatically updated to reflect the change. The purpose is to reduce redundant work by combining the documentation task together with the configuration task.

The solution presented achieved this by interconnecting the software with the use of a Python script. The requirements were fulfilled, and therefore the request is met. However, the requirements only consisted of the purpose of the system and components to be used, it left out any specification on the extent of the automation. It is unclear as to whether the developed solution was enough to satisfy Robert Anderssons needs. Ideally Robert Andersson should have been contacted and ask about the specifications.

During the analysis phase of this work, we found out an alternative solution to the problem. A software with the name of Nornir can provide similar, if not the same functionalities as Ansible. Nornir is entirely managed by Python code, and could have been used instead of the somewhat diffusing Ansible Python API in the developed Python script. Another alternative solution to achieve network automation of device managed in NetBox, could potentially be done by exploring the development of a new NetBox plugin. This would add the functionality to the NetBox server without being dependent on other software and systems. The request specifically stated that Ansible was to be used as the configuration engine, so the alternative solutions was not explored any further. Alternative solutions or adding additional configuration support for the script to automate, as well as optimizing the workflow can be explored in a future work.

The methodology followed a logical structure. First an understanding of the required components was established, then different designs were explored using several flowcharts. When the design was agreed upon, a pseudo code was developed and after that the development of the script began. The work would have benefited from conducting an active elicitation of the functional specifications with the requestor of the solution, as mentioned above. Also, the testing phase should have been conducted outside of the development environment with feedback from actual users of the system. A feedback session with Robert Andersson should have been done to confirm whether the developed solution was as envisioned. This, as well as not well defining requirements and a concrete problem formulation, which also could have been established by leveraging Robert Anderssons expertise, has made the end product somewhat difficult to evaluate in some regard. However it is uncertain if Robert Andersson had any better defined requirements or a concrete problem formulation in mind. Considering this, establishing such a framework with very little information and insight first of all would be understandably difficult. The evaluation is repeatable and objective though, one only needs to follow the manual of the product and test the solution against devices with the same capabilities and resources needed that are mentioned in this thesis.

6 Conclusion

This work provides a solution for achieving network automation based on the combined usage of Ansible, NetBox and RESTCONF. A python script was developed to allow interconnection of the components.

When a user navigates the NetBox GUI, and makes a change to a networking device by either updating the hostname, create/update/delete an interface or assign/remove an IP-address to an interface, the device will be configured to reflect the change and to save the new configuration. The change in the documentation is mediated to the script by a webhook. The script parses the webhook data and decide whether to act on the change or not, if the change is deemed to be configurable, it will create a suitable configuration in a RESTCONF supported format. Ansible is then invoked to handle the communication to the device with the configuration as payload. The configuration is being sent as RESTCONF.

The solution was developed targeting a Cisco CSR1000v router, running IOS-XE. No other networking devices was a part of the development nor the testing phase. This results in limited functional support for other types of devices, as RESTCONF depends on existing YANG data models for interpretation of the configuration received. The change of hostname and configuration saving targets a Cisco only data model, support for these are therefor only available for Cisco devices.

This work can serve as a first iteration as a part of a DevOps approach. Automation support for more configurations can be extended in a future work, as well as adding support for more network devices and added functionality.

The main problem has been answered. It was possible to build a script that engages with the predefined tools and create a solution to configure network devices and automate the actual configuration. The solution as it is now is not very widely applicable, so at this moment it most likely will not improve anyone's situation. However, the developed script is released as open-source to make it available for further development or implementation. In the end, the solution is deemed to have accomplished the request for a network automated system, although with some limitations. Moreover, a feedback session with the requester was never conducted to verify this. The solution is not exactly new, but it appears that it is somewhat unique. Generally other tools have been used to accomplish the same functionality as this solution. Although this might not be entirely true since one circumstance of a forum poster was observed claiming to have used similar tools and achieve the same results. This claim however is never backed up by a script or any type of evidence. So, by lack of substance behind this claim it at least appears that the solution presented in this paper is unique, but the concept is not new.

References

[1] Michael O'Brien, 2020-06-18, *Using Netbox Webhooks to Update Cisco Devices.* https://journey2theccie.wordpress.com/2020/06/18/using-netbox-webhooks-to-update-cisco-devices/ [Accessed 2021-11-29]

[2] John McGovern, 2020-09-29, *Nornir vs Ansible: Which Automation Tool Is Better?.* https://www.youtube.com/watch?v=uQAaA_-lb7k [Accessed 2021-11-29]

[3] Bengt Rundqvist, *automatisering.* http://www.ne.se/uppslagsverk/encyklopedi/lång/automatisering [Accessed 2021-11-29]

[4] Macarthy R. W. and Bass J. M., 2020-08-26 – 2020-08-28, *An Empirical Taxonomy of DevOps in Practice.* [2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)]

[5] Dyck A., Penners R. and Lichter H., 2020-05-19, *Towards Definitions for Release Engineering and DevOps,* [2015 IEEE/ACM 3rd International Workshop on Release Engineering]

[6] NetBox Documentation, *What is NetBox?,* https://netbox.readthedocs.io, [Accessed 2021-11-29]

[7] Red Hat Ansible web site, *Red Hat Ansible*, https://ansible.com [Accessed 2021-11-29]

[8] Cowan J., Fang A., Grosso P., Lanz K., Marcy G., Thompson H., Tobin R., Veillard D., Walsh N. and Yergeau F. 2008-11-26, *Extensible Markup Language (XML) 1.0 (Fifth Edition),* https://www.w3.org/TR/2008/REC-xml-20081126/ [Accessed 2021-11-29]

[9] Fuller A., Fan Z., Day C. and Barlow C. 2020-06-23, *Digital Twin: Enabling Technologies, Challenges and Open Research,* [School of Computing and Mathematics, Keele University][Astec IT Solutions Ltd.]

[10] The CentOS Project, *CentOS Linux EOL.* https://centos.org/centos-linux-eol/ [accessed 2021-11-29]

[11] Pallets Projects, User's Guide v2.0.x, *Foreword.* https://flask.palletsprojects.com/en/2.0.x/foreword/ [accessed 2021-11-29]

[12] Ganesh Nalawade, Ansible 4 documentation, collection index. *ansible.netcommon.restconf_config – Handles create, update, read and delete of configuration data on RESTCONF enabled devices.* https://docs.ansible.com/ansible/latest/collections/ansible/netcommon/restconf_config_module.html [accessed 2021-11-29]

[13] Ansible project contribiutors, Ansible 4 documentation, developer's guide, *Python API.* https://docs.ansible.com/ansible/latest/dev_guide/developing_api.html [accessed 2021-11-29]

Appendix A: Flowcharts

This appendix contains the different flowcharts that was developed as potential design options. Fig. A, displays the flowchart that was selected as the initial design goal and which the pseudocode is based on. The rest of the flowcharts features ideas that were explored but ultimately not chosen as the design.

Fig. B, describes a potential design idea that aims to provide a more user-friendly experience with the intent of an interface that prompts the user to enter user-specific data to be used in the script (potentially using a NetBox plugin). Which would allow the user to manage configuration settings without accessing the source code of the script. Fig. C, displays a flowchart a similar goal as the latter, but instead aims to use a file to manage the configuration that is to be supported. Fig. D, explored a possible functionality that aims to automatically import the network devices into NetBox's database when IP-addresses are created in NetBox. The feature would leverage a custom field where the user could enable it or disable it, when sending the webhook for the created IP-address. The script would then loop through the addresses, either by searching the Ansible inventory file or conducting an active search on the network using a RESTCONF supported query, collecting information about the device if there was a match. If the device doesn't exist in the NetBox database, it was to be created by using the NetBox API.

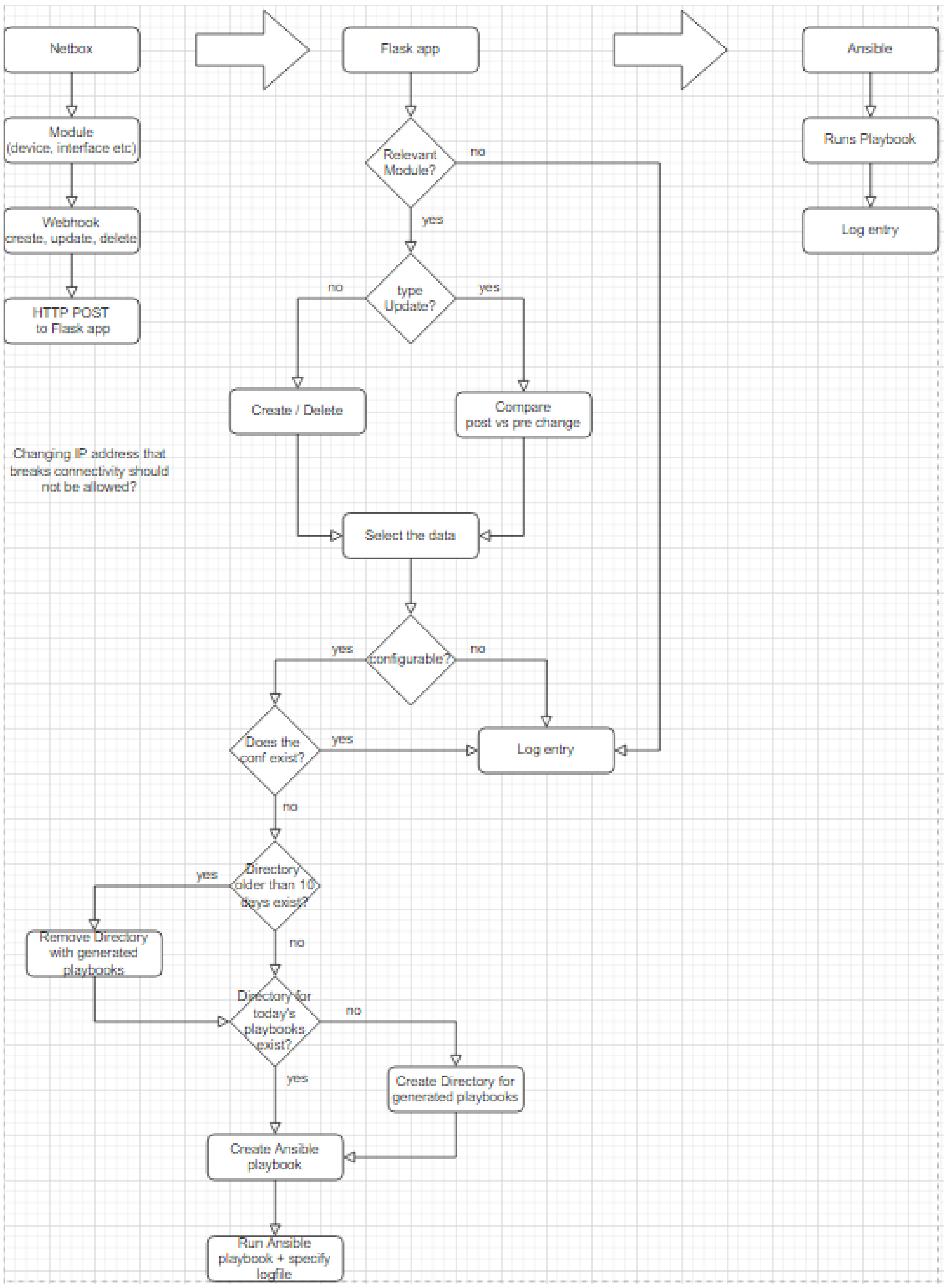

Netbox
Module (device, interface etc)
Webhook create, update, delete
HTTP POST to Flask app
Flask app
Relevant Module?
no
yes
type Update?
no
yes
Create / Delete
Compare post vs pre change
Changing IP address that breaks connectivity should not be allowed?
Select the data
configurable?
yes
no
Does the conf exist?
yes
Log entry
no
Directory older than 10 days exist?
yes
Remove Directory with generated playbooks
no
Directory for today's playbooks exist?
no
Create Directory for generated playbooks
yes
Create Ansible playbook
Run Ansible playbook + specify logfile
Ansible
Runs Playbook
Log entry

Fig. A. The flowchart that was selected as the initial design goal, displayed in more detail.

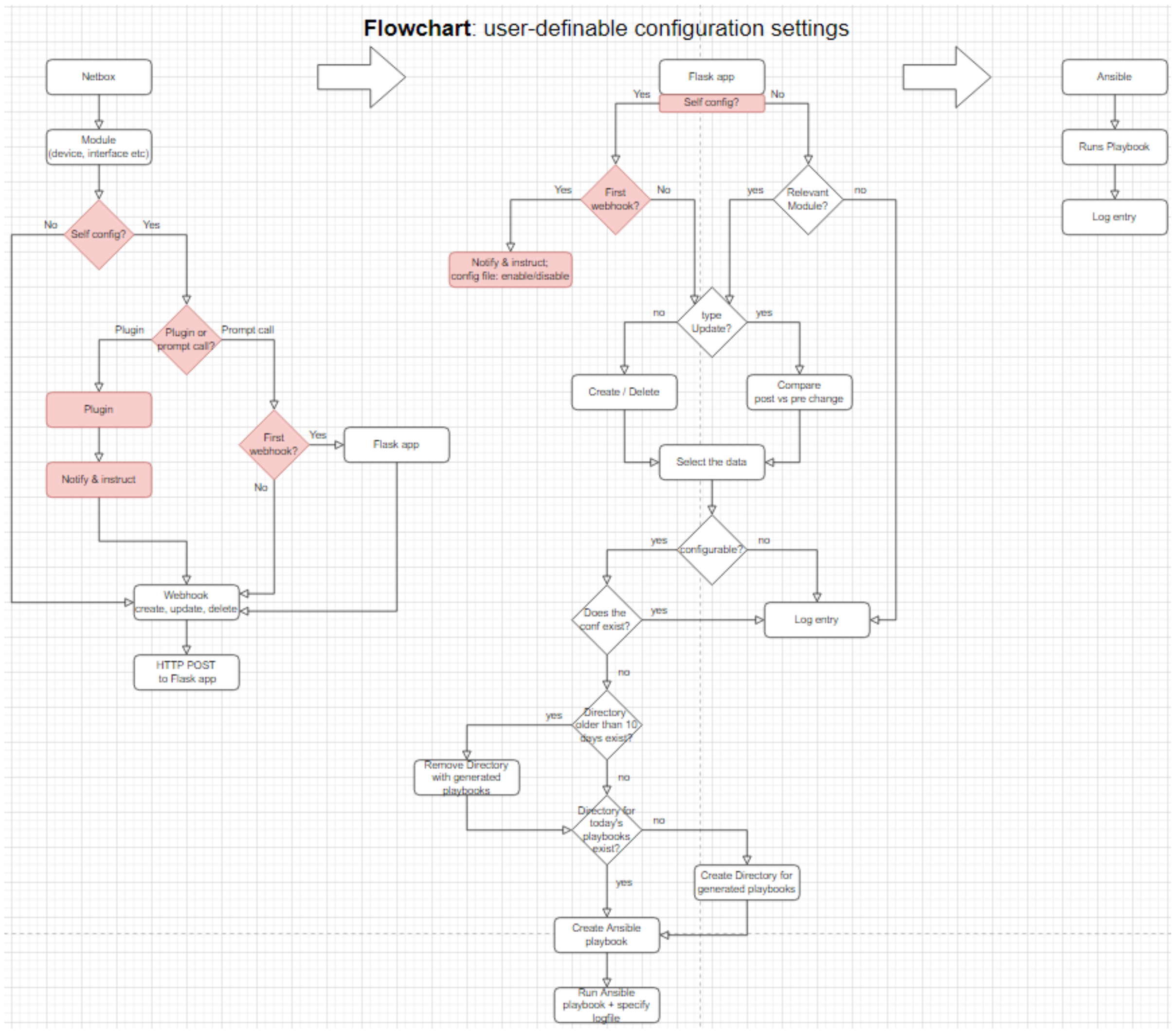
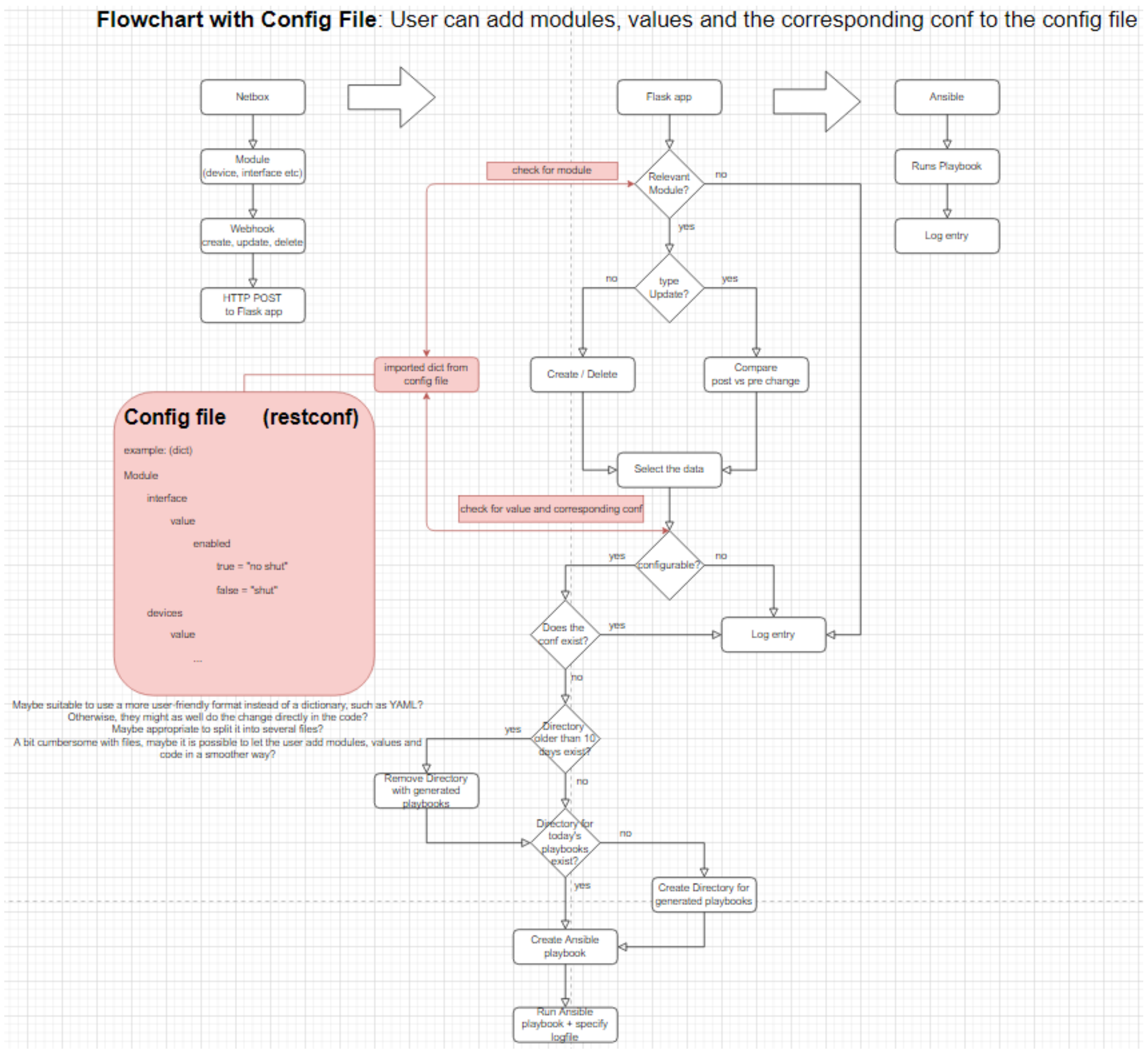

Flowchart: user-definable configuration settings
Netbox
Module (device, interface etc)
Self config?
No
Yes
Plugin or prompt call?
Plugin
Prompt call
Plugin
Notify & instruct
First webhook?
Yes
No
Flask app
Webhook create, update, delete
HTTP POST to Flask app
Flask app
Self config?
Yes
No
First webhook?
Yes
No
Notify & instruct; config file: enable/disable
Relevant Module?
yes
no
type Update?
no
yes
Create / Delete
Compare post vs pre change
Select the data
configurable?
yes
no
Does the conf exist?
yes
no
Log entry
Directory older than 10 days exist?
yes
no
Remove Directory with generated playbooks
Directory for today's playbooks exist?
no
yes
Create Directory for generated playbooks
Create Ansible playbook
Run Ansible playbook + specify logfile
Ansible
Runs Playbook
Log entry

Fig. B. A flowchart depicting a more user-firendly interaction with the script by utilizing a graphical interface on the NetBox server.


Flowchart with Config File: User can add modules, values and the corresponding conf to the config file

Maybe suitable to use a more user-friendly format instead of a dictionary, such as YAML?
Otherwise, they might as well do the change directly in the code?
Maybe appropriate to split it into several files?
A bit cumbersome with files, maybe it is possible to let the user add modules, values and code in a smoother way?

Netbox
Module (device, interface etc)
Webhook create, update, delete
HTTP POST to Flask app
Flask app
Ansible
Runs Playbook
Log entry
check for module
Relevant Module?
no
yes
type Update?
no
yes
Create / Delete
Compare post vs pre change
imported dict from config file
Config file (restconf)
example: (dict)
Module
interface
value
enabled
true = "no shut"
false = "shut"
devices
value
...
Select the data
check for value and corresponding conf
configurable?
yes
no
Does the conf exist?
yes
Log entry
no
Directory older than 10 days exist?
yes
Remove Directory with generated playbooks
no
Directory for today's playbooks exist?
no
Create Directory for generated playbooks
yes
Create Ansible playbook
Run Ansible playbook + specify logfile

Fig. C. A flowchart that illustrates the design of using a configuration file to allow the user to add and remove configuration that will be sent as RESTCONF.

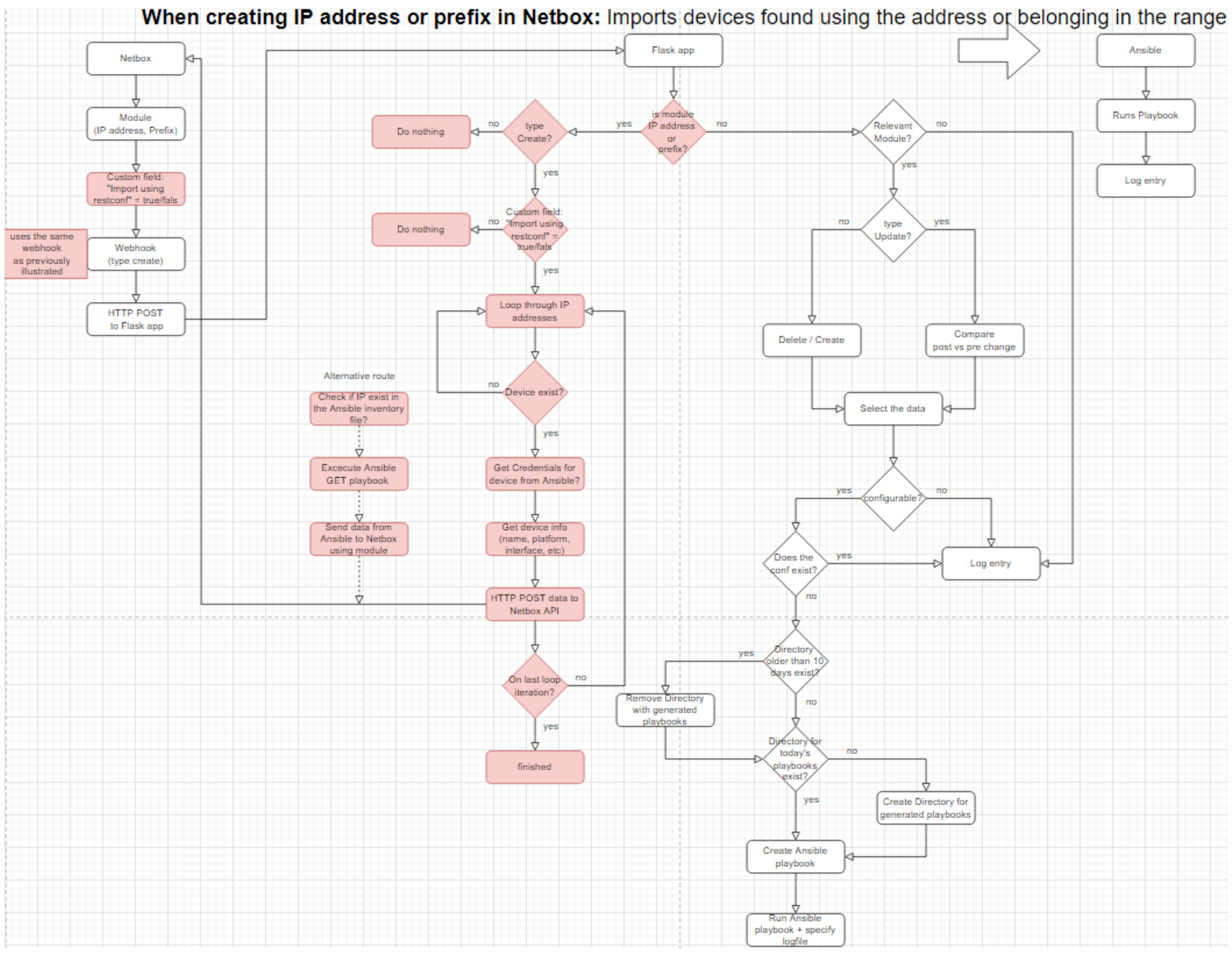

When creating IP address or prefix in Netbox: Imports devices found using the address or belonging in the range
Netbox
Module (IP address, Prefix)
Custom field: "Import using restconf" = true/fals
uses the same webhook as previously illustrated
Webhook (type create)
HTTP POST to Flask app
Flask app
is module IP address or prefix?
yes
no
type Create?
Do nothing
Custom field: "Import using restconf" = true/fals
Do nothing
Loop through IP addresses
Device exist?
Alternative route
Check if IP exist in the Ansible inventory file?
Excecute Ansible GET playbook
Send data from Ansible to Netbox using module
Get Credentials for device from Ansible?
Get device info (name, platform, interface, etc)
HTTP POST data to Netbox API
On last loop iteration?
finished
Relevant Module?
type Update?
Delete / Create
Compare post vs pre change
Select the data
configurable?
Does the conf exist?
Log entry
Directory older than 10 days exist?
Remove Directory with generated playbooks
Directory for today's playbooks exist?
Create Directory for generated playbooks
Create Ansible playbook
Run Ansible playbook + specify logfile
Ansible
Runs Playbook
Log entry

Fig. D. A flowchart that descripes a feature for automatically importing network devices into the NetBox database, when a device uses an IP-address was added to NetBox and if the device doesn't already exist in the database. The device would be created by utilizing the NetBox API.

Appendix B: Pseudocode

This appendix contains the pseudocode that was derived from the intial flowchart, which was used as the design goal. The pseudocode aided the developers but was only used as a general guideline when building the python script. Some functionality described below were never added or was altered in the finished product. Functionalities that had a lower implementation pritorty were not always finilized in how they were to operate.

Handle webhook message from Netbox

```
1. Script receives webhook

2. Script tests IP reachability to intended configuration interface (TCP handshake?)
   (add later)
    2.1. If reachable = true then continue
    2.2. If reachable = false then correct Netbox through api? Information message?
         Reject webhook and, and log occurrence?

3. Compare model against a list containing configurable modules by calling the function
    3.1. If model = true then continue
    3.2. if model = false then call log function

4. Compare event
    4.1. If event = update then pick out the differences between post vs pre change
            4.1.1. Send the difference as in-data to "pick out values" function
    4.2. If event != update then send all values as in-data to "pick out values" function

5. "Pick out values" function: searches the dict and returns the configurable values
   (model specific function?)
    5.1. Pick values
            5.1.1. If a value = true then append to a new dict or something?
            5.1.2. If all value = false then call log function
    5.2. Send values to "Create playbook" function
```

Get Credentials from netbox by api (might be added later)

Session-key is needed for decryption of the secrets password that is stored in netbox

```
1. Maybe: generate RSA private-key to use in the session-key request?
2. Get request to receive session-key
```

3. Get request to receive decrypted secrets for the specific device with session-key attached in header.
4. The secret is sent to the playbook function in order to be used in the playbook

Create playbook

1. Runs to see if conf exists already by calling conf exist function? Or check for existing playbooks? Maybe added later
2. Runs Clean up playbook function? Maybe added later
3. Create playbook
    - 3.1. Create file
    - 3.2. create payload in JSON

Config exists? (might be added later)

1. Information in file? Playbook? Device?
2. Config exist = true?
    - 2.1. Config exist = partial or different?
        - 2.1.1. Warning message/options? Compare conf? Override conf (user responsibility)?
3. If config exist = false: push conf

Clean up playbooks (might be added later)

1. Check for directory with datestamp older than x days
    - 1.1. If true = removes the dir with the playbooks inside
    - 1.2. If false = do nothing
        - 1.2.1. If true
        - 1.2.2. If false

Run Ansible with the playbook and specify log file

1. Execute Ansible with arg(playbook x, y, z)
2. Get response
    - 2.1. If successful = true: message or do nothing?
    - 2.2. If successful = false: message? Try again? Options?
    - 2.3. If successful = true but network/operation impact: update Netbox to reflect impact
3. Produce logfile

Log function

1. Prints the in-data in the log file together with date, device and user

List of webhook models:

- Interface
- IP-address
- Device

List of event types:

- Created
- Updated
- Deleted

Points in user manual

- User must appoint one address on device as primary IP. If both an IPv6 address and an IPv4 address are appointed as primary addresses, then both addresses will be show up as “IP4 primary” & “IP6 primary” respectively, but only one address can be displayed as “primary IP”. The IPv6 address will be prioritized for if both are present. If you would like to have an IPv4 address as primary address you cannot have an IPv6 address as primary at the same time.

Appendix C: Python script

The source code for the python script is available on GitHub for public use. The open-source license and manual can also be accessed here. Repository destination; https://github.com/albiriku/OmniConf. The source code for the script can also be inspected below, although the formatting of the code is not ideally presented in Word. For a more suitable format, please vist the repository.

```
"""
        OmniConf - used with Netbox and Ansible to automate certain device
    configuration using restconf
        Copyright (C) 2021, Alexander Birgersson & Rickard Kutsomihas.
        This program is free software: you can redistribute it and/or mod-
    ify
        it under the terms of the GNU General Public License as published
    by
        the Free Software Foundation, either version 3 of the License, or
        (at your option) any later version.
        This program is distributed in the hope that it will be useful,
        but WITHOUT ANY WARRANTY; without even the implied warranty of
        MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
        GNU General Public License for more details.
        You should have received a copy of the GNU General Public License
        along with this program.  If not, see <http://www.gnu.org/li-
    censes/>.
    """


    #!/usr/bin/env python


    # the imports are needed for Ansible API (to run playbooks)
    from __future__ import (absolute_import, division, print_function)
    __metaclass__ = type


    import json
    import shutil


    import ansible.constants as C
    from ansible.executor.task_queue_manager import TaskQueueManager
    from ansible.module_utils.common.collections import ImmutableDict
    from ansible.inventory.manager import InventoryManager
    from ansible.parsing.dataloader import DataLoader
    from ansible.playbook.play import Play
    from ansible.plugins.callback import CallbackBase
    from ansible.vars.manager import VariableManager
    from ansible import context


    # the imports needed for flask and HTTP requests
    from flask import Flask, request, Response              # used for
    flask app, receive and response of webhook
```

```
import requests                                         # used for
HTTP get request to netbox api
from requests.auth import HTTPBasicAuth                 # used for
creating the basic authentication field in the HTTP header
app = Flask(__name__)


# IMPORTANT: YOUR user specific settings:
FLASK_PATH = '/webhook-test'
# the path that flask listens to for webhooks, which should also be
appended to url webhook destination
NETBOX_IP = 'https://193.10.237.252'
# ip address to netbox
NETBOX_TOKEN = 'Token c788f875f6a0bce55f485051a61dbb67edba0994'
# user token to be able to communicate with netbox api
ANSIBLE_INVFILE = '/home/albiriku/devnet/dne-dna-code/intro-ansi-
ble/hosts'  # path to Ansible inventory file
ANSIBLE_VAULTPASS = 'secret'
# ansible vault password for decryption


# "configurable" contains the values from the webhook we deem are con-
figurationable for the corresponding model
# "informational" contains additional information required for config-
uration
list_of_models =    {
                    'device':
                                {
                                    'configurable': ['name'],
                                    'informational': ['primary_ip',
'address']
                                },
                    'interface':
                                {
                                    'configurable': ['name', 'type',
'enabled', 'description'],
                                    'informational': ['device', 'url']
                                },
                    'ipaddress':
                                {
                                    'configurable': ['address'],
                                    'informational': ['assigned_ob-
ject', 'device', 'url']
                                },
                    }




# check if "model" is configurable and returns "True" if match
def check_model(model):
    if model in list_of_models:
        # match
        print(model, 'is configurable')
        return True
```

```
# used when event is "updated"
# returns the difference of "prechange" vs "postchange" in a new dict
def compare(prechange, postchange):
    updated_values = {}
    # loops through keys and non-matching values
    for key in prechange:
        if prechange[key] != postchange[key]:
            updated_values[key] = postchange[key]
    return updated_values




# returns the configurable values
def pick_out_values(model, data, values):
    """
    This function consists of 2 parts.
    Part 1:
    Loops through the elements in "configurable" in "list_of_models",
    then compares the elements to the elements in "values".
    If the compared values match the values are added to the
    dict "configuration".
    Part 2:
    Either populates "information" with a url to the device in Netbox,
    or a primary ip address if the model is "device".
    The url is needed to retrieve the primary ip address to the device
    and the primary ip address is required in order to send the con-
figuration.
    Returns "config" as a dict with both "configurable" and "informa-
tional"
    or returns a value of "None".
    """


    # part 1
    # creates a new dict
    config = {}
    # adds a nestled dict to hold the configuration values
    config['configuration'] = {}


    # loop comparing the elements in "configurable" and "values"
    for element in list_of_models[model]['configurable']:
        if element in values:
            # if matched value is an empty string the loop continues
with the next iteration
            if values[element] == '':
                continue
            # key and value is added in the dictionary "configuration"
            config['configuration'][element] = values[element]


        # in order to update the devices hostname when a primary ip
gets assigned in netbox
        # the key "name" and its value from "data" is added to the
configuration dict
        elif 'primary_ip6' in values or 'primary_ip4' in values:
            config['configuration'][element] = data[element]


    # part 2
    # executes only if "configuration" is populated
    if config['configuration'] != {}:
        # creates a second dict in "config", named "information"
        config['information'] = {}
        # the content of "informational" is added to "info"
        info = list_of_models[model]['informational']
        # the number of elements present in "info"
        length = len(info)


        for i in range(0, length):#loop på antalet element i info
           # during the first iteration
            if i == 0:
                # the first key in "information" is added
                config['information'] = data[info[i]]
                # if the key value is equal to "None" the function
ends and returns "None"
                if config['information'] == None:
                    return None


            # performs iterative lookups and overwrites the value in
order to perform subsequent lookups in the nestled dicts
            # the element in "info" is matched and replaces key:value
pair in "information"
            # "information" ends up with the key corresponding the
last element in "info" along with its lookup value
            else:
                config['information'] = config['information'][info[i]]
        return config
    # when no configuration values matched
    return None




# performs a HTTP GET request to netbox api for the devices' primary
ip address
def get_api_data(config):
    # consist of the ip address to netbox and the url to the device
    url = NETBOX_IP + config['information']
    # HTTP header
    headers =   {
                'Content-Type': 'application/json',
                'Authorization': NETBOX_TOKEN
                }
    # performs the GET request
    api_data = requests.request('GET', url, headers=headers, ver-
ify=False)
    # response data as json
    api_data = api_data.json()
    # returns the primary ip address if its present on the device
    # otherwise returns "None"
    if api_data['primary_ip'] != None:
        ip = api_data['primary_ip']['address']
        return ip
    else:
        return None



# this function separates prefix and address from each other
# returns both or only address depending on parameters given
def split_address(address, mask=False):
    if address[-2] == '/':
        #prefix is 1 digit
        prefix = address[-1:]
        address = address[:-2]
    elif address[-3] == '/':
        #prefix is 2 digits
        prefix = address[-2:]
        address = address[:-3]
    else:
        #for ipv6 only, when prefix is 3 digits
        prefix = address[-3:]
        address = address[:-4]
    if mask == False:
        return address
    else:
        return address, prefix




# this class is taken from the Ansible python API example:
"https://docs.ansible.com/ansible/latest/dev_guide/develop-
ing_api.html"
# create a callback plugin so we can capture the output
class ResultsCollectorJSONCallback(CallbackBase):
    """A sample callback plugin used for performing an action as re-
sults come in.
    If you want to collect all results into a single object for pro-
cessing at
    the end of the execution, look into utilizing the ``json``
callback plugin
    or writing your own custom callback plugin.
    """


    def __init__(self, *args, **kwargs):
        super(ResultsCollectorJSONCallback, self).__init__(*args,
**kwargs)
        self.host_ok = {}
        self.host_unreachable = {}
        self.host_failed = {}


    def v2_runner_on_unreachable(self, result):
        host = result._host
        self.host_unreachable[host.get_name()] = result


    def v2_runner_on_ok(self, result, *args, **kwargs):
        """Print a json representation of the result.
        Also, store the result in an instance attribute for retrieval
later
        """
        host = result._host
        self.host_ok[host.get_name()] = result
        print(json.dumps({host.name: result._result}, indent=4))


    def v2_runner_on_failed(self, result, *args, **kwargs):
        host = result._host
        self.host_failed[host.get_name()] = result


# this code is also taken from "https://docs.ansible.com/ansible/lat-
est/dev_guide/developing_api.html"
# but it has been heavily edited
# this function creates and runs an Ansible play
def run_playbook(config, ip, event, model, data, prechange):
    """
    Part 1:
    Apart from "host" this part consist of the orignal code,
    although some parameters have been changed as well.
    Initializes and loads necessary data for Ansible to operate.
    Part 2:
    Creates the data structure that represents our play, including
tasks, this is basically what our YAML loader does internally.
    The Ansible restconf_config module is used for each play (task),
which yang data model used might differ between the plays.
    The task and payload is defined using prior extrapolated data
where the task that will be performed by the device is decided by the
event,
    i.e. a "deleted" event will result in a "delete" task being exe-
cuted. Appropriate parameters for the event are also specified in the
task,
    such as "path" or "method". The payload consists of the appropri-
ate yang-data-model used and the configuration dict.
    Part 3:
    Compiles and executes the Ansible playbook and reports back the
result.
    Part 4:
    Saves the configuration on the device to startup-config.
    This doesnt seem to be doable with the ansible restconf plugin us-
ing the cisco-ia module,
    so we use requests to send a HTTP post msg instead of executing it
as a playbook.
    """


    # part 1
    # removes mask from the ip
    host = split_address(ip)
    # since the API is constructed for CLI it expects certain options
to always be set in the context object
    context.CLIARGS = ImmutableDict(connection='smart', forks=10, ver-
bosity=True, check=False, diff=False)


    # initialize needed objects
    loader = DataLoader() # takes care of finding and reading yaml,
json and ini files
    passwords = dict(vault_pass=ANSIBLE_VAULTPASS)


    # instantiate our ResultsCollectorJSONCallback for handling re-
sults as they come in. Ansible expects this to be one of its main dis-
play outlets
    results_callback = ResultsCollectorJSONCallback()


    # create inventory, use path to host config file as source or
hosts in a comma separated string
    inventory = InventoryManager(loader=loader, sources=ANSI-
BLE_INVFILE)


    # variable manager takes care of merging all the different sources
to give you a unified view of variables available in each context
    variable_manager = VariableManager(loader=loader, inventory=inven-
tory)


    # instantiate task queue manager, which takes care of forking and
setting up all objects to iterate over host list and tasks
    # IMPORTANT: This also adds library dirs paths to the module
loader
    # IMPORTANT: and so it must be initialized before calling
`Play.load()`.
    tqm = TaskQueueManager(
        inventory=inventory,
        variable_manager=variable_manager,
        loader=loader,
        passwords=passwords,
        stdout_callback=results_callback,  # use our custom callback
instead of the ``default`` callback plugin, which prints to stdout
    )


    # part 2
    # interface configuration
    # uses the ietf-yang-data-model interfaces-module
    if model == 'interface':
        # name of the target interface
        # used in the path when altering an existing interface
        name = data['name']
        # first checks if 'type' exist in conf, then converts the in-
terface type
```

```
        # from a netbox value to a value supported by the ietf-inter-
face module
        if 'type' in config['configuration']:
            # "virtual" gets converted to "softwareLoopback"
            if config['configuration']['type'] == 'virtual':
                config['configuration']['type'] = 'softwareLoopback'
            # other types gets converted to "ethernetCsmacd"
            else:
                config['configuration']['type'] = 'ethernetCsmacd'


        # when interface is created in netbox
        if event == 'created':
            # "configuration" dict as payload
            payload = {"ietf-interfaces:interface":config['configura-
tion']}
            # this task will create the interface on the device
            task = [dict(action=dict(module='ansible.netcom-
mon.restconf_config', args=dict(path='/data/ietf-interfaces:interfac-
es', content=json.dumps(payload), method='post')))]


        # when interface is edited in netbox
        elif event == 'updated':
            payload = {"ietf-interfaces:interface:":config['configura-
tion']}
            # updates the interface on the device
            task = [dict(action=dict(module='ansible.netcom-
mon.restconf_config', args=dict(path=f'/data/ietf-interfaces:inter-
faces/interface={name}', content=json.dumps(payload),
method='patch')))]


        # when interface is deleted in netbox
        elif event == 'deleted':
            # deletes the interface on the device
            task = [dict(action=dict(module='ansible.netcom-
mon.restconf_config', args=dict(path=f'/data/ietf-interfaces:inter-
faces/interface={name}', method='delete')))]


    # ipaddress configuration
    if model == 'ipaddress':
        # the target interface
        name = data['assigned_object']['name']
        # ipv4 or ipv6
        family = data['family']['label'].lower()
        # ip address to conf
        address = config['configuration']['address']


        # ip and mask needs to be separated for the payload format
        address, prefix = split_address(address, True)


        # only needed for IPv4, converts prefix to netmask format
        if family == 'ipv4':
            # reference: https://stackoverflow.com/ques-
tions/23352028/how-to-convert-a-cidr-prefix-to-a-dotted-quad-netmask-
in-python
            prefix = '.'.join([str((m>>(3-i)*8)&0xff) for i,m in enu-
merate([-1<<(32-int(prefix))]*4)])
            mask = 'netmask'


        elif family == 'ipv6':
            mask = 'prefix-length'


        # payload & task for IPv4 & IPv6
        if event == 'created' or event == 'updated':


            # payload structure same for created and updated
            payload =   { 'ietf-interfaces:interface':
                            { f'ietf-ip:{family}':
                                { 'address':
                                    [{
                                        'ip': address,
                                        mask: prefix
                                    }]
                                }
                            }
                        }


            if event == 'created':
                # adds the new address to the interface
                task = [dict(action=dict(module='ansible.netcom-
mon.restconf_config', args=dict(path=f'/data/ietf-interfaces:inter-
faces/interface={name}',
                        content=json.dumps(payload),
method='patch')))]


            elif event == 'updated':
                # the target object on the device
                old_address = split_address(prechange['address'])


                # first deletes the old address then adds the new ad-
dress
                task = [dict(action=dict(module='ansible.netcom-
mon.restconf_config', args=dict(path=f'/data/ietf-interfaces:inter-
faces/interface={name}/ietf-ip:{family}/address={old_address}',
                        method='delete'))),
                        dict(action=dict(module='ansible.netcom-
mon.restconf_config', args=dict(path=f'/data/ietf-interfaces:inter-
faces/interface={name}',
                        content=json.dumps(payload),
method='patch')))]


        elif event == 'deleted':
            # deletes the address
            task = [dict(action=dict(module='ansible.netcom-
mon.restconf_config', args=dict(path=f'/data/ietf-interfaces:inter-
faces/interface={name}/ietf-ip:{family}/address={address}',
method='delete')))]


    # device configuration (hostname only)
    # uses the Cisco IOS XE native yang data model
    if model == 'device':
        # during device creation in netbox no primary ip address is
associated with the device
        # therefore hostname can only be configured on the physical
device via an update
        # hostname will be updated on the device when a primary ip
gets assigned as well
        if event == 'updated':
            hostname = config['configuration']['name']
            # blank space not supported as a part of the devices'
hostname
            hostname = hostname.replace(" ", "-")
            payload = {'Cisco-IOS-XE-native:hostname': f'{hostname}'}


            task = [dict(action=dict(module='ansible.netcom-
mon.restconf_config', args=dict(path='/data/Cisco-IOS-XE-native:na-
tive/hostname',
                        content=json.dumps(payload),
method='patch')))]


    # part 3
    # generates and runs the playbook with the task given above
    play_source = dict(
    name='Ansible Play',
    hosts=[host],
    gather_facts='no',
    tasks=task
    )


    # Create play object, playbook objects use .load instead of init
or new methods,
    # this will also automatically create the task objects from the
info provided in play_source
    play = Play().load(play_source, variable_manager=variable_manager,
loader=loader)


    # Actually run it
    try:
        result = tqm.run(play)  # most interesting data for a play is
actually sent to the callback's methods
    finally:
        # we always need to cleanup child procs and the structures we
use to communicate with them
        tqm.cleanup()
```

```
            if loader:
                loader.cleanup_all_tmp_files()


        # Remove ansible tmpdir
        shutil.rmtree(C.DEFAULT_LOCAL_TMP, True)


        # Prints the outcome of playbook that executed
        print('SUCCESSFUL ***********')
        for host, result in results_callback.host_ok.items():
            # when a playbook performs delete
            if not 'candidate' in result._result:
                print('{0} >>> {1} \n{2}'.format(host, result._re-
sult['changed'], result._result['invocation']))
            # when a playbook performs create/update
            else:
                print('{0} >>> {1}'.format(host, result._result['candi-
date']))


        print('FAILED *******')
        # failed to execute the play
        for host, result in results_callback.host_failed.items():
            print('{0} >>> {1}'.format(host, result._result['msg']))


        print('UNREACHABLE *********')
        # couldnt reach the host
        for host, result in results_callback.host_unreachable.items():
            print('{0} >>> {1}'.format(host, result._result['msg']))


        # part 4
        # saves the configuration on the device
        # loaded_vars contains all the host variables that ansible loads
from the varfiles
        loaded_vars = variable_manager._hostvars
        # restconf username loaded from ansible
        username = loaded_vars[host]['ansible_user']
        # restconf password loaded from ansible
        password = loaded_vars[host]['ansible_httpapi_password']


        # uses cisco-ai module to invoke an RPC that saves the running
conf to startup
        path = 'https://' + host + '/restconf/operations/cisco-ia:save-
config'
        header =  {'Content-type': 'application/yang-data+json'}
        # creates a HTTP basic auth field with the restconf user/password
        dev_auth = HTTPBasicAuth(username, password)


        # sends the HTTP post
        saveconf = requests.post(path, headers=header, verify=False,
auth=dev_auth)
        # saves the response msg
```

```
    saveconf = saveconf.json()
    # prints the response msg
    print(json.dumps(saveconf, indent=4))




# the parameters which flask listens to for webhooks
@app.route(FLASK_PATH, methods=['POST'])
def respond():
    """
    This functions runs when receiving a webhook.
    Below is the flask app code that receives the webhook.
    Calls the functions responsible for each step.
    If a function returns a value of None, the webhook wont be pro-
cessed futher.
    In the code this is represented as a "return Response(=200)",
which is a
    HTTP response to the HTTP webhook POST.
    Functions and terms are further explained in conjunction with the
functions

    First of all the received webhook is stored in a variable and
    some of its' important parts are stored in other variables.
    These variables are used in different functions to extract data.
    Step 1 calls the function "check_model" which uses the "model"
    part of the webhook in order to check if the model is configura-
ble.
    This is arbitrarily predetermined in the dict "list_of_models".
    If model is determined to be configurable the "event" value of the
webhook
    is saved to be processed in step 2. Undetermined configurability
will end the process.
    Step 2 compares the different events in the webhook. If event is
"updated" then the function "compare()"
    is called, which compares the "prechange" part of the webhook with
the "postchange" part.
    The differences between the pre- and postchange are saved in the
"values" variable.
    If the event is "created" the "postchange" is saved in "values".
    If the event is "deleted" the "prechange" is saved in "values".
    Step 3 configurable values are selected by calling the
"pick_out_values()" function with the arguments
    "data", "model" and "values". The function will also return a url
to a device in Netbox when "model" is not "device".
    If no configurable values can be selected, the process end.
    Step 4 depending on what model is being configured the retrieval
of ip address differ. If the model is "device" the
    address is included in the webhook, otherwise the url to the de-
vice has to be used and then via Netbox API retrieve the
    address from the device. If the device doesnt have a primary ip
address assigned to it, the process will end.
    Step 5 is the last step. Calls the "run_playbook()" function in
order to create and run an Ansible playbook with data from previous
steps.
```

```
    The configuration will be saved on the device. The function only
offers full support for Cisco IOS XE devices. For other devices sup-
port might vary.
    """


    # the webhook payload is stored in "webhook"
    webhook = request.json
    # the model which the webhook originated from
    model = webhook['model']
    # the data portion of the webhook
    data = webhook['data']
    # post- and prechange information
    prechange = webhook['snapshots']['prechange']
    postchange = webhook['snapshots']['postchange']


    print(json.dumps(webhook, indent = 4))


    # step 1: check if model is configurable
    if check_model(model) == True:
        event = webhook['event']


    # if model is not configurable
    else:
        # ends
        print('model not configurable')
        return Response(status=200)


    # step 2: check event
    if event == 'updated':
        if prechange == None:
            # when prechange contains a value of null
            # this occurs when the "make this the primary IP for the
device" option was changed when creating/editing an ipaddress
            # which will send a device webhook as well, with the pre-
change set to null
            return Response(status=200)

        else:
            values = compare(prechange, postchange)


    elif event == 'created':
        values = postchange


    elif event == 'deleted':
        if prechange == None:
            # when interface is deleted, netbox sends a delete webhook
for the ipaddress as well, with empty post- and prechange.
            # in this case the address will be removed along with the
interface on the device, which mean no futher action is needed.
            return Response(status=200)
        else:
            values = prechange


    #step 3: get configurable values and api url if more info needed
    config = pick_out_values(model, data, values)
    print('Configurable values: ', config)


    if config == None:
        # no configurable values were returned from the "pick_out_val-
ues" function
        # this can also happen when:
        # 1. the device has no primary IP assigned to it, or
        # 2. a webhook is triggered for an ipaddress which has not
been assigned to a device, assigned_objects contains a value of null
        # in this case no configuration has been made, because the
change doesnt relate to a device
        return Response(status=200)


    #step 4: api get request to retrive device primary IP address (in-
cluded in the webhook for device model)
    if model == 'device':
        ip = config['information']


    else:
        ip = get_api_data(config)
        print('device primary IP is', ip)
        if ip == None:
            print()
            print('The targeted device has no primary IP assigned.
Nowhere to send conf.')
            return Response(status=200)


    #step 5: create and run playbook
    run_playbook(config, ip, event, model, data, prechange)

    return Response(status=200)
```