Wi-Fi automation lab

Lab Guide - Day 2

Andreas Koksrud / Telenor Bedrift

Co-authors:

Kjetil Teigen Hansen & François Vergès

telenor

References / inspiration

https://github.com/CiscoDevNet/yangsuite/
https://postman.com
https://docs.ansible.com/ansible/latest/collections/cisco/ios/ios_facts_module.html
https://www.wifireference.com/2020/01/14/viewing-network-telemetry-from-the-catalyst-9800-with-grafana/
https://grafana.com/grafana/dashboards/13462-device-health-monitoring/
https://grafana.com/grafana/dashboards/12468-catalyst-9800-client-stats/
https://wirelessisfun.wordpress.com/2020/12/10/network-telemetry-data-and-grafana-part-1-the-advanced-netconf-explorer/
https://python.org
https://codeium.com
https://canonical.com/multipass
https://blog.apnic.net/

Copyright

Any slides marked Kjetil Teigen Hansen or the Conscia logo will also have full or co-ownership by Kjetil

Any slides marked François Vergès or the SemFio logo will also have full or co-ownership by François

Prerequisites

- Cisco Meraki account (https://dashboard.meraki.com)
- Juniper MIST account (https://manage.mist.com)
- Postman account (https://postman.com)
- Complete the pre-lab exercises (this document) before the deep dive labs
- Bring an Ethernet dongle if you don't have built-in port
- (optional) Bring an extra screen to show lab guide (or prepare to Alt-Tab)

Communications

- WebEx space: Wi-Fi automation lab
- Please help each other
- Sharing is caring ☺

Agenda

Pre-lab

- Choose your hypervisor
- Install Ubuntu Server w/Docker
- Install Postman
- Install VS Code
- (optional) install 9800-CL

Day 1

- Sort out pre-lab task problems
- Get to know the lab environment
- Connect VS Code to Ubuntu
- Install and explore Ansible
- Explore Python automation
- Install and explore YANG Suite
- Explore Postman
- Install and explore Grafana

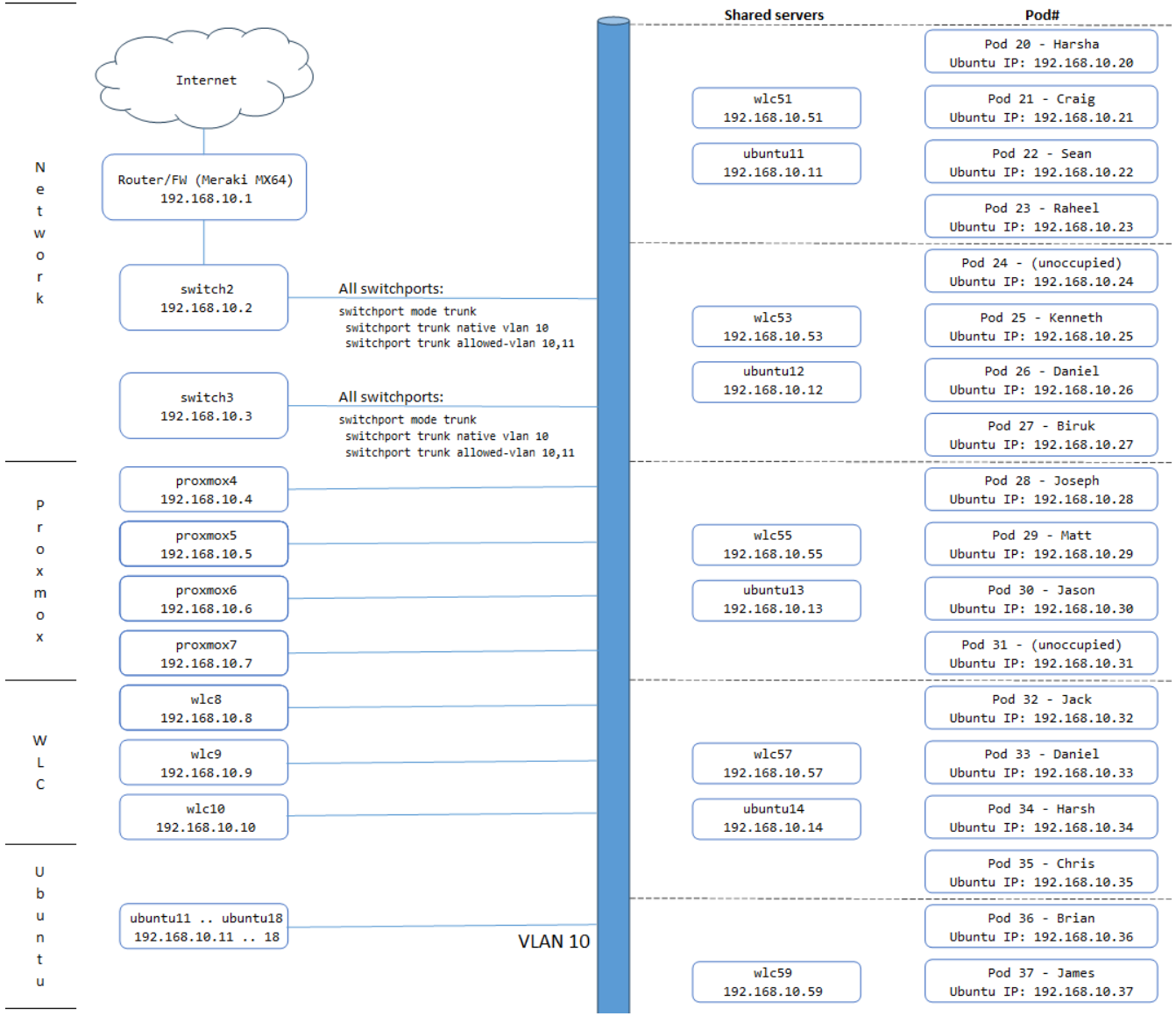
Day 2

- In-depth explore a topic of choice
  - Grafana / TIG-stack
  - Grafana Cloud
  - Ansible
  - Python
  - Cloud vendor automation (MIST or Meraki)

Scope

- In scope
  - Getting started with various systems/languages/solutions for lab purposes
  - Set up your own Ubuntu Linux server on your own laptop. It will be possible to use a shared server if you do not want or have possibility to install an Ubuntu VM on your laptop
  - Some nice examples to try various aspects of automation
  - Inspiring you to explore deeper on your own
- Out of scope topics
  - Git
  - Learning the languages (Ansible, Python, InfluxQL, etc)
  - Learning Linux
  - Deploying the systems for production use
  - Troubleshooting WLC/AP connection

Wi-Fi Automation deep dive - Lab topology



- Each student have an assigned Pod number. When using static IP, their Ubuntu Server should have the same last octet as the Pod#
- 2 students share a preconfigured WLC, as assigned in the topology map
- Everyone is connected to VLAN 10
- All switchports on all switches are trunk ports with VLAN 10 as native, and allowed vlan as VLAN 10 and 11
- For Wi-Fi clients on your SSIDs you can use VLAN 11 to not fill up VLAN 10

Login to shared devices

User: devnet-adm
Pass: ChangeMe2025!

IP Plan (VLAN 10)

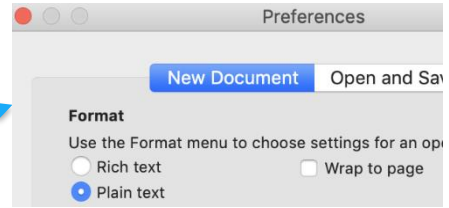
Static adm IPs: 192.168.10.1 - 19
Ubuntus (per pod): 192.168.10.20 - 50
WLCs (shared): 192.168.10.51 - 70
DHCP range: 192.168.10.71 - 250

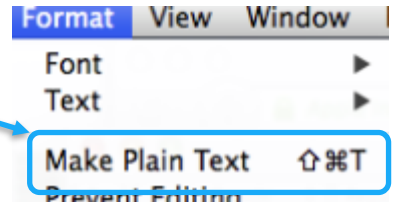
IP Plan (VLAN 11)

Static adm IPs: 192.168.11.1 - 10
DHCP range: 192.168.10.11 - 250

Notes for Mac users - TextEdit

- TextEdit (the default text editor on OS X) use rich text formatting by default. This is not a great idea when copy-pasting stuff to the command line, so there is a couple of things you must keep in mind
  - If you want, you can change a setting in TextEdit to make all new documents plain text
  - To change the current document to plain text go to Format -> Make Plain Text
  - Shortcut is: Shift-Cmd-T





About Day 2

- In-depth exploration of one or more topics of your choice
- You should have completed relevant parts of Pre-lab tasks and Day 1 exercises
- For each exercise, we will try to do the following
  1. Tasks
     - One or more slides, with one or more tasks
     - Expected output
  2. Hints
     - One or more slides, with hints or code snippets to get you in the right direction
  3. Example solution
     - An example solution
       (Note that most exercises will have many possible solutions that are better than the example ☺)
     - Example output
- Error handling
  - Examples in this lab will have minimal error handling to keep the examples small and readable
  - Depending on the usage (one-off vs. a recurring operations script) you will want some error handling

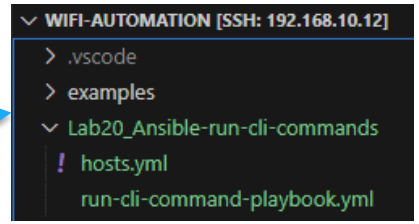
In-depth: Ansible

- This part of the Day 2 labs will focus on using Ansible
- The following lab exercises are included in this part
  - Lab Exercise #20: Ansible - Run CLI commands
  - Lab Exercise #21: Ansible - CLI configuration
  - Lab Exercise #22: Ansible - Using Jinja2 templates
  - Lab Exercise #23: Ansible - Using RESTCONF
  - Lab Exercise #24: Ansible - Writing your own module using Python
  - Lab Exercise #25: Ansible - Using RESTCONF + Python-module
  - Lab Exercise #26: Ansible - Organizing projects using import_playbook
  - Lab Exercise #27: Ansible - Organizing projects using roles

Run a CLI command - Intro

- In this exercise we will write a playbook that runs a single command on our WLC
- We will observe that the output of the CLI command is passed back to the playbook, and can be used in further tasks
- For this exercise, the output will be used in a task that prints the result of the CLI command out to the terminal where you ran the playbook

Run a CLI command - Tasks



- Task 0: Set up the files and folders and VS Code
  - Create a working directory
  - Create a playbook file
  - Remember to activate the automation-venv

Python 3.12.3 (ansible-venv)

```
devnet-adm@ubuntu-devnet:~/wifi-automation/Lab20_Ansible-run-cli-commands$ source ~/automation-
venv/bin/activate
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab20_Ansible-run-cli-commands$ ansible --version
```

- Task 1: Write a playbook that
  - Runs the command "show wireless summary | inc Max APs"
  - Output when running this on the WLC should be like this

```
wlc9# show wireless summary | inc Max APs
Max APs supported                : 100
```

- Task 2: Run the playbook on your WLC
  - You can copy the inventory file hosts.yml from Day 1

hosts.yml

```
wlc:
  hosts:
    192.168.10.{WLC-IP}:
  vars:
    ansible_connection: network_cli
    ansible_network_os: ios
    ansible_ssh_pass: restconf-pass
    ansible_password: restconf-pass
    ansible_user: restconf-adm
    ansible_host_key_checking: False
```


Run a CLI command - Hints

- The structure of the file should be like this

run-cli-command-playbook.yml

```
---
- name: CLI Playbook
  hosts: wlc
  connection: network_cli
  gather_facts: false

  tasks:

    - name: "Get Max APs supported with CLI command"
      some_module_that_will_run_commands:
        some_parameter: "show wireless summary | include Max APs"
      register: result

    - name: "View result"
      some_builtin_module_that_can_output_messages:
        some_parameter: "{{ result.stdout }}"
```

- Run the playbook with this command

```
(automation-venv) (.cut.)$ ansible-playbook -i hosts.yml run-cli-command-
playbook.yml
```

- Playbook header
  - name: Name of the playbook, make it descriptive
  - hosts: The "wlc" here you can find in the hosts.yml file. Can be expanded to other types.
  - connection: network_cli for most of our stuff
  - gather_facts: A general-purpose gathering function
- Our "tasks" section starts here
- Instead of this placeholder, you would use a module that can run CLI commands. Try looking in the cisco.ios collection in the Ansible Docs
  https://docs.ansible.com/ansible/latest/collections/cisco/ios/index.html#plugins-in-cisco-ios
- Instead of this placeholder, you will use a module that can output stuff. The example solution uses a builtin module to output the text in "results" as a debug message when you run the playbook. Try looking in the ansible.builtin collection
  https://docs.ansible.com/ansible/latest/collections/ansible/builtin/index.html#plugins-in-ansible-builtin

Run a CLI command - Example solution

- Here is an example solution
- The output should look like this

cli-playbook.yaml

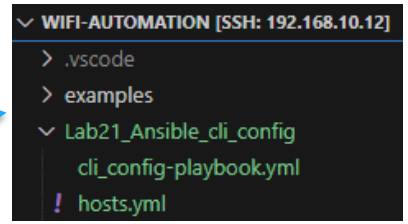
```
---
- name: CLI Playbook
  hosts: wlc
  connection: network_cli
  gather_facts: false

  tasks:

    - name: "Get Max APs supported with CLI command"
      cisco.ios.ios_command:
        commands: "show wireless summary | include Max APs"
      register: result

    - name: "View result"
      ansible.builtin.debug:
        msg: "{{ result.stdout }}"
```

```
(ansible-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab20_Ansible-run-cli-commands$ ansible-playbook -i hosts.yml run-cli-command-playbook.yml

PLAY [CLI Playbook] ************************************************************************************************************************************

TASK [Get Max APs supported with CLI command] **********************************************************************************************************
ok: [192.168.10.10]

TASK [View result] *************************************************************************************************************************************
ok: [192.168.10.10] => {
    "msg": [
        "Max APs supported          : 1000"
    ]
}

PLAY RECAP *********************************************************************************************************************************************
192.168.10.10              : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

(ansible-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab20_Ansible-run-cli-commands$
```


Ansible - CLI configuration - Intro

- In this exercise, we will write a playbook that does some CLI configuration
- A key concept of Ansible is "idempotency". That basically means
  - If something IT NOT configured, then go ahead and configure it
  - If something IS configured, just skip the task.
  - That means: DO NOT configure the same stuff again if it isn't necessary
- This exercise will create a task that does this for a small part of the configuration. It can of course be expanded to a complete config. There are many ways to do this, some of which are
  - Putting everything in a gigantic playbook with hundreds of tasks (it might get heavy to navigate)
  - Splitting tasks for small config snippets in separate files, and call them all from a "master playbook"
  - Using Jinja2 templates (we will try this in a later exercise)
  - Using Ansible roles in combination with templates or tasks
  - ... and many many more ;-)

Ansible - CLI configuration - Tasks



- Task 0: Set up the files and folders and VS Code
  - Create a working directory
  - Create a playbook file
  - Remember to activate the automation-ve

Python 3.12.3 (ansible-venv)

```
devnet-adm@ubuntu-devnet:~/wifi-automation/Lab21_Ansible_cli_config$ source ~/automation-venv/bin/activate
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab21_Ansible_cli_config$ ansible --version
```

- Task 1: Write a playbook that
  - Ensure that this policy profile is present on the WLC

```
wireless profile policy clients_policy_profile
 idle-timeout 3000
 passive-client
 session-timeout 43200
 vlan 11
 no shutdown
```

- Task 2: Run the playbook on your WLC
  - You can copy the inventory file hosts.yml from Day 1
- Task 3: Verify the idempotency of the playbook. That means
  - If the config IS NOT present, it should be configured
  - If the config IS present, it should do nothing

hosts.yml

```
wlc:
  hosts:
    192.168.10.{WLC-IP}:
  vars:
    ansible_connection: network_cli
    ansible_network_os: ios
    ansible_ssh_pass: restconf-pass
    ansible_password: restconf-pass
    ansible_user: restconf-adm
    ansible_host_key_checking: False
```


Ansible - CLI configuration - Hints

- The structure of the file should be like this

cli_config-playbook.yml

```yaml
---
- name: CLI Playbook
  hosts: wlc
  connection: network_cli
  gather_facts: false

  tasks:

    - name: "Configure policy profile: clients_policy_profile"
      some_module_that_will_do_ios_config:
        some_parameter_with_config_lines:
          - idle-timeout 3000
          - passive-client
          - session-timeout 43200
          - vlan 11
          - no shutdown
        some_parent_parameter: wireless profile policy clients_policy_profile
```

- Instead of this placeholder, you would use a module that can do CLI config on Cisco IOS devices. Try looking in the cisco.ios collection in the Ansible Docs
  https://docs.ansible.com/ansible/latest/collections/cisco/ios/index.html#plugins-in-cisco-ios
- The module needs all the config lines that we want, as an ordered list, after a parameter specifying all those lines
- The module also needs a parameter specifying which parent the config lines should belong to. The parent in this example is the policy profile. In another example it might be an interface, a WLAN, a site tag etc.

- Run the playbook with this command

```
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab21_Ansible_cli_config$ ansible-playbook -i hosts.yml cli-config-playbook.yml
```


Ansible - CLI configuration - Hints 2

- If you try to change the VLAN ID from 11 to 12, you will notice that nothing changes on the WLC. That is because the policy profile needs to be shutdown before making changes if it already exists. You can test this on your WLC with the following commands

```
wlc9# wlc9#conf t
wlc9(config)# wireless profile policy clients_policy_profile
wlc9(config-wireless-policy)# vlan 12
% Policy profile needs to be disabled before performing this operation.
```

- To address this, we can use the `before:` and `after:` statements in the task, to get Ansible to add certain commands before and after the lines that we compare. The before and after sections will only be run if the task itself would create a change (i.e. if the lines do not match)

(parts of) cli_config-playbook.yml

```
- name: "Configure policy profile: clients_policy_profile"
      cisco.ios.ios_config:
        before:
          - wireless profile policy clients_policy_profile
          - shutdown
        lines:
          - idle-timeout 3000
          - passive-client
          - session-timeout 43200
          - vlan 11
        after:
          - no shutdown
        some_parent_parameter: wireless profile policy clients_policy_profile
```

- These lines will be entered before the lines that we compare
- Since we enter "no shutdown" after the lines themselves, we do not need this in the lines
- These lines will be entered after the lines that we compare

Ansible - CLI configuration - Example solution

- Here is an example solution
- First time run, output should be like this, and the policy profile appear on WLC

```
(ansible-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab21_Ansible_cli_config$ ansible-playbook -i hosts.yml cli_config-playbook.yml

PLAY [CLI configuration] ********************************************************************************************************************

TASK [Configure policy profile: clients_policy_profile] *************************************************************************************
[WARNING]: To ensure idempotency and correct diff the input configuration lines should be similar to how they appear if present in the runn
on device
changed: [192.168.10.10]

PLAY RECAP **********************************************************************************************************************************
192.168.10.10              : ok=1    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

- On next run, the config should already be present

```
(ansible-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab21_Ansible_cli_config$ ansible-playbook -i hosts.yml cli_config-playbook.yml

PLAY [CLI configuration] ********************************************************************************************************************

TASK [Configure policy profile: clients_policy_profile] *************************************************************************************
ok: [192.168.10.10]

PLAY RECAP **********************************************************************************************************************************
192.168.10.10              : ok=1    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

cli-playbook.yaml

```
---
- name: CLI configuration
  hosts: wlc
  connection: network_cli
  gather_facts: false

  tasks:

    - name: "Configure policy profile: clients_policy_profile"
      cisco.ios.ios_config:
        before:
          - wireless profile policy clients_policy_profile
          - shutdown
        lines:
          - idle-timeout 3000
          - passive-client
          - session-timeout 43200
          - vlan 11
        after:
          - no shutdown
        parents: wireless profile policy clients_policy_profile
```

```
wlc9800#show run | sec wireless profile policy
wireless profile policy lab-policy
 no central authentication
 no central dhcp
 no central switching
 dhcp-tlv-caching
 http-tlv-caching
 vlan 11
 no shutdown
wireless profile policy clients_policy_profile
 idle-timeout 3000
 passive-client
 session-timeout 43200
 vlan 11
 no shutdown
wireless profile policy default-policy-profile
 description "default policy profile"
```


Ansible - CLI configuration - Extra tasks

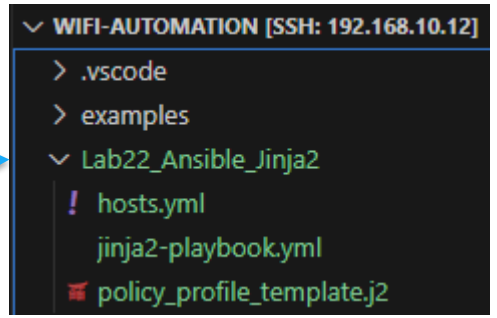
- Create some more tasks in the playbook, for configuring
  - Interfaces. You can play with Gig2 and Gig3, or create VLAN interfaces or Loopbacks
  - WLANs. PSKs might be tricky to get idempotent once they are encrypted.
  - Global commands, like ensure restonf is enabled
- Remember, for idempotency the commands must be exactly as they will appear in the running config. So don't use abbreviations in the config lines (like "desc" for description, "ip add" for ip address, etc.
  - They will be run and work, but next time you run they will be configured again since the command to push is different from what is present on the device.
- The before and after statemens will have to be changed accordingly. For tasks that do not need to be shutdown, you will probably not need the before and after sections

Using Jinja2 templates - Intro

- Jinja2 is a templating language, widely used by developers to create text files that have placeholders, which can be filled inn with various values. One very common example is HTML
- Jinja2 is a native part of Ansible and the YAML files there. So anytime you write something in double curly braces it is a Jinja2 variable that will be replaced by the content of the variable when you run the playbook
- Instead of using Jinja2 inline, we can store templates as separate files, and then refer to those templates when running our playbook
- In this exercise, we will modify the playbook from our CLI configuration exercise to use Jinja2 templates instead of spelling out the commands in the playbook itself
- One advantage of using templates is to reuse a playbook for various templates
- Another advantage is the possibility to loop in the template itself, for instance looping over all site tags

Using Jinja2 templates - Tasks

- Task 0: Set up the files and folders and VS Code



  - Create a working directory
  - Create a playbook file
  - Remember to activate the automation-ve

Python 3.12.3 (ansible-venv)

```
devnet-adm@ubuntu-devnet:~/wifi-automation/Lab22_Ansible_Jinja2$ source ~/automation-venv/bin/activate
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab22_Ansible_Jinja2$ ansible --version
```

- Task 1: Write a playbook using a Jinja2 template

```
wireless profile policy clients_policy_profile_j2
session-timeout 43200
 vlan 11
 no shutdown
```

  - Ensure that this policy profile is present on the WLC
  - You can copy the inventory file hosts.yml from Day 1

hosts.yml

```
wlc:
  hosts:
    192.168.10.{WLC-IP}:
  vars:
    ansible_connection: network_cli
    ansible_network_os: ios
    ansible_ssh_pass: restconf-pass
    ansible_password: restconf-pass
    ansible_user: restconf-adm
    ansible_host_key_checking: False
```

  - Verify the idempotency (changes stuff only when needed)
- Task 2: Change some of the text in the j2 template
  - Use a variable for the name and the VLAN ID
  - Use a loop statement in the task to do multiple policy profiles with the same template

Using Jinja2 templates - Hints for task 1

- The structure of the file should be like this

jinja2-playbook.yml

```yaml
---
- name: CLI Playbook
  hosts: wlc
  connection: network_cli
  gather_facts: false

  tasks:

    - name: "Configure policy profile: clients_policy_profile_j2"
      cisco.ios.ios_config:
        before:
          - wireless profile policy clients_policy_profile_j2
          - shutdown
        src: policy_profile_template.j2
        match: line
        after:
          - no shutdown
```

This is where we put in the Jinja2 template file we created

Hint: It should contain the commands that you want in the config, as they appear in the config

policy_profile_template.j2

```
wireless profile policy clients_policy_profile_j2
 session-timeout 43200
 vlan 11
 no shutdown
```

- Now, go ahead and fill inn the "policy_profile_template.j2" file
- Run the playbook with this command

```
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab22_Ansible_Jinja2$ ansible-playbook -i hosts.yml jinja2-playbook.yml
```


Using Jinja2 templates - Hints for task 2

- Now we will try changing some of the static config in the template with some parameters
- The parameters can come from different sources
  - hosts.yml (will be used in this exercise)
  - group_vars/ and host_vars/ directories
  - External systems using APIs (NetBox, Catalyst Center, etc)
- Let us modify our files
  - hosts.yml
  - jinja2-playbook.yml
  - policy_profile_template.j2

Create some variables in the bottom of the hosts.yml file

```
ansible_host_key_checking: False
policy_profiles:
  - name: policy_profile_j2
    vlan: 11
  - name: policy_profile_j2_2
    vlan: 12
  - name: policy_profile_j2_3
    vlan: 13
```

Use the "loop" statement in the playbook, to loop over the policy_profiles list you created.
- To reference the items that you loop over, you use keyword "item"
- To reference sub-values, use dot notation or square bracket notation.

Examples of both here, to reference the name and vlan of each item

```
- name: "Configure policy profile from template: {{ item.name }}"
  cisco.ios.ios_config:
    before:
      - wireless profile policy {{ item['name'] }}
      - shutdown
    src: policy_profile_template.j2
    match: line
    after:
      - no shutdown
  loop: "{{ policy_profiles }}"
```

Use the notation in the .j2 template as well

```
wireless profile policy {{ item['name'] }}
 session-timeout 43200
 vlan {{ item['vlan'] }}
 no shutdown
```


Using Jinja2 templates - Example solution

Change to YOUR WLC IP

jinja2-playbook.yml

```yaml
---
- name: CLI configuration using Jinja2 template
  hosts: wlc
  connection: network_cli
  gather_facts: false

  tasks:

    - name: "Configure policy profile from template: {{ item.name }}"
      cisco.ios.ios_config:
        before:
          - wireless profile policy {{ item['name'] }}
          - shutdown
        src: policy_profile_template.j2
        match: line
        after:
          - no shutdown
      loop: "{{ policy_profiles }}"
```

hosts.yml

```yaml
wlc:
  hosts:
    192.168.10.9:
  vars:
    ansible_connection: network_cli
    ansible_network_os: ios
    ansible_ssh_pass: ChangeMe2025!
    ansible_password: ChangeMe2025!
    ansible_user: devnet-adm
    ansible_host_key_checking: False
    policy_profiles:
      - name: policy_profile_j2
        vlan: 11
      - name: policy_profile_j2_2
        vlan: 12
      - name: policy_profile_j2_3
        vlan: 13
```

policy_profile_template.j2

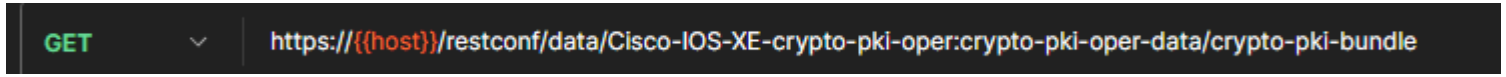
```
wireless profile policy {{ item['name'] }}
 session-timeout 43200
 vlan {{ item['vlan'] }}
 no shutdown
```

- Optional tasks:
  - Can you change more parameters (ex. session-timeout value)?
  - Can you create templates for other stuff? Loopback interfaces (safe to play with), wlans, site tags, etc?

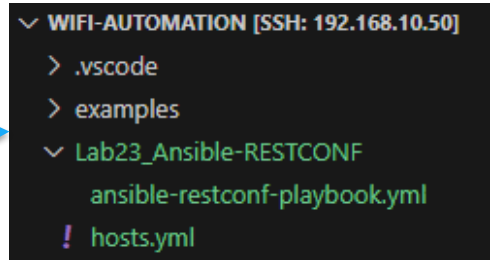
Ansible - Using RESTCONF - Intro

- We can do RESTCONF calls directly from Ansible
- Some pros with RESTCONF instead of CLI plugins from Ansible
  - The return can be JSON, which is easy to use like a dictionary
  - The speed of getting back data is high, especially if you compare to larger amount of CLI data. For instance, getting a table of connected clients when you have a lot of them
- Some cons with RESTCONF instead of CLI plugins from Ansible
  - There must exist a YANG model for what you are looking for
  - You must find the correct xpath of the YANG model

- In this exercise, we will get the certificates on the WLC, and more specifically the validity dates associated with the certs
- We will use the YANG module "Cisco-IOS-XE-crypto-pki-oper" to get operational data about the certificates



Ansible - Using RESTCONF - Tasks



- Task 0: Set up the files and folders and VS Code
  - Create a working directory
  - Create a playbook file
  - Copy the hosts.yml from previous projects
  - Remember to activate the automation-venv

Python 3.12.3 (ansible-venv)

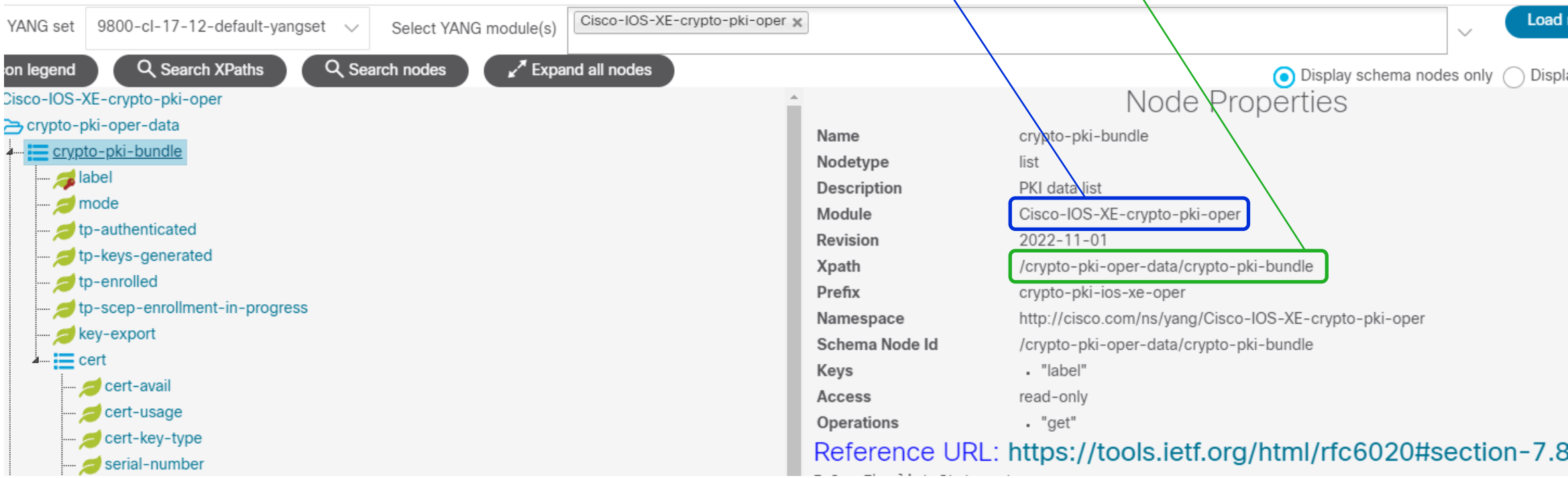
```
devnet-adm@ubuntu-devnet:~/wifi-automation/Lab23_Ansible_RESTCONF$ source ~/automation-venv/bin/activate
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab23_Ansible_RESTCONF$ ansible --version
```

- Task 1: Use YANG Suite (or yangcatalog.org) to find the RESTCONF path that will give details about the certificates
  - Load and view the details of the module "Cisco-IOS-XE-crypto-pki-oper"
  - Test the RESTCONF call towards your WLC with Postman
  - Add some filters, to only return the label, subject-name and validity-end
- Task 2: Write the Ansible task to use your RESTCONF path
  - Use the "ansible.builtin.uri" module to perform the RESTCONF call
  - Print the results of the call
  - (optional task) Print the results of just the validity date of each result

Ansible - Using RESTCONF - Hints for Task 1

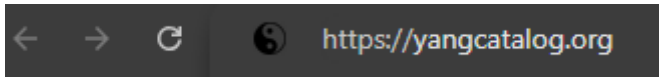
- Example on using YANG Suite to browse the "Cisco-IOS-XE-crypto-pki-oper" model
  - RESTCONF call for 9800: https://{WLC_IP}/restconf/data/{YANG module}:{xpath}
    WLC_IP: 192.168.10.9 (change to YOUR WLC IP)
    YANG module: Cisco-IOS-XE-crypto-pki-oper
    Xpath: crypto-pki-oper-data/crypto-pki-bundle

```
https://{{host}}/restconf/data/Cisco-IOS-XE-crypto-pki-oper:crypto-pki-oper-data/crypto-pki-bundle
```



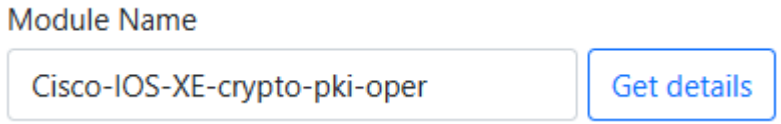
Ansible - Using RESTCONF - Hints for Task 1

- Example on using yangcatalog.org to browse the "Cisco-IOS-XE-crypto-pki-oper" model
  - Use your browser to enter https://yangcatalog.org

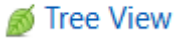


  - Click "YANG Module Detail Viewer"

  - Start writing and then select the module. Click "Get details"



  - Click "Tree View"



  - Decide which level of data you want to retrieve. Let's start with the full bundle. You can click each line to expand or compact levels, or click the + next to "Element" to expand the full tree **Element**
    - On the right hand of the "crypto-pki-bundle" line you will find the path you need in the "Sensor Path" column

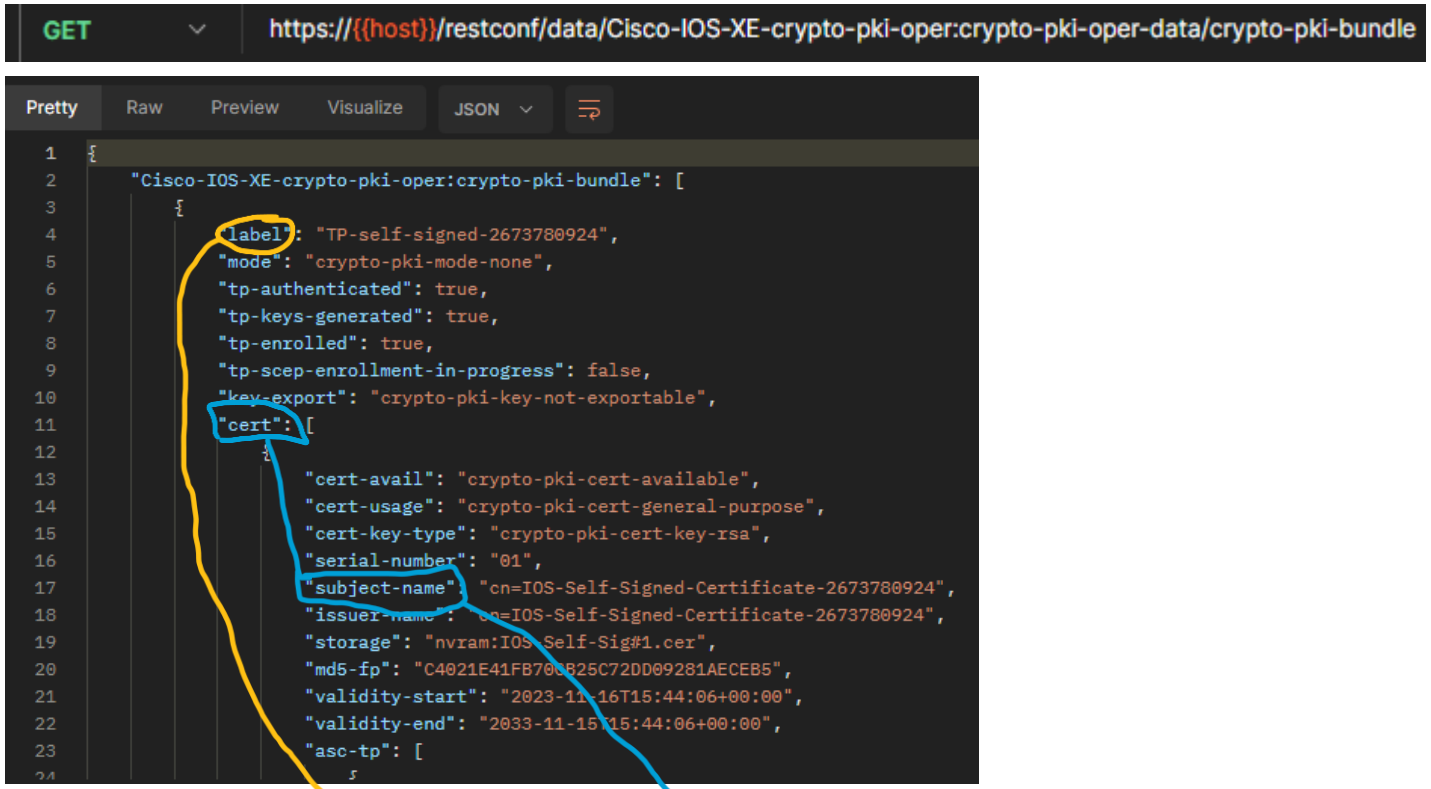
Element	Schema	Type	Flags	Opts	Status	Path	Sensor Path
Cisco-IOS-XE-crypto-pki-oper	module	module					
crypto-pki-oper-data	container	container	no config		current	/crypto-pki-ios-xe-oper:crypto-pki-oper-data	/Cisco-IOS-XE-crypto-pki-oper:crypto-pki-oper-data
crypto-pki-bundle	list	list	no config		current	/crypto-pki-ios-xe-oper:crypto-pki-oper-data/crypto-pki-ios-xe-oper:crypto-pki-bundle	/Cisco-IOS-XE-crypto-pki-oper:crypto-pki-oper-data/crypto-pki-bundle

  - Copy the sensor path, and create the RESTCONF call:

```
https://{{host}}/restconf/data/Cisco-IOS-XE-crypto-pki-oper:crypto-pki-oper-data/crypto-pki-bundle
```


Ansible - Using RESTCONF - Hints for Task 1

- Enter the path in Postman



- You should hopefully get something like this, where you can see lots of details for each certificate on the device

- To choose only a few fields, use the operator "?fields=" after the path. This example shows picking the label, subject-name and validity-end from above

```
https://{{host}}/restconf/data/Cisco-IOS-XE-crypto-pki-oper:crypto-pki-oper-data/crypto-pki-bundle?fields=label;cert/subject-name;cert/validity-end
```


Ansible - Using RESTCONF - Hints for Task 2

- Here are some tips for writing the Ansible playbook
- To perform the RESTCONF call, use the module "ansible.builtin.uri". Check the Ansible documentation for how-to and examples
- "url" should be copy-paste of what you have tested in Postman, just replace {{host}} with {{ ansible_host}}
- Try to replicate headers, body format and method from Postman
- Do not validate certs
- Specify some valid HTTP status codes, at least status code 200
- Create a second task for printing the results. Start by printing all of the result.
- Try to narrow down the printing (think, what can you do with the results beside printing to the screen. It might be out of scope for this lab, but very nice to test how you can manipulate the results). Tips, msg: {{result['json']}} is the first step
- Use a when: statement on the print results part. It should only run when the status code is 200 "OK".

Ansible - Using RESTCONF - Example solution

- Playbook and task header is similar to previous tasks
- Using ansible.builtin.uri
- URL copy-paste from what is tested in Postman
- Change host, user and password with the ansible_host, ansible_user and ansible_password values
- Try to replicate method, headers and body format from Postman
- Do not validate the HTTPS certs (at least in this lab)
- Valid return status codes
- View the results using ansible.builtin.debug module
- Only when the HTTP status code of the RESTCONF call is 200 ("OK")

```yaml
---

- name: RESTCONF demo playbook
  hosts: wlc
  connection: network_cli
  gather_facts: false

  tasks:

    - name: "Get Certificate list with RESTCONF"
      ansible.builtin.uri:
        url: "https://{{ ansible_host }}/restconf/data/Cisco-IOS-XE-crypto-pki-oper:cr
        user: "{{ ansible_user }}"
        password: "{{ ansible_password }}"
        method: GET
        headers:
          Content-Type: 'application/yang-data+json'
          Accept: 'application/yang-data+json'
        body_format: json
        body:
        validate_certs: false
        status_code:
          - 200
          - 204
          - 404
      register: result

    - name: "View result"
      ansible.builtin.debug:
        msg: "{{ result['json']['Cisco-IOS-XE-crypto-pki-oper:crypto-pki-bundle'] }}"
      when: result.status == 200
```

ansible-restconf-playbook.yml

```yaml
---

- name: RESTCONF demo playbook
  hosts: wlc
  connection: network_cli
  gather_facts: false

  tasks:

    - name: "Get Certificate list with RESTCONF"
      ansible.builtin.uri:
        url: "https://{{ ansible_host }}/restconf/data/Cisco-IOS-XE-crypto-pki-
oper:crypto-pki-oper-data/crypto-pki-bundle?fields=label;cert/subject-
name;cert/validity-end"
        user: "{{ ansible_user }}"
        password: "{{ ansible_password }}"
        method: GET
        headers:
          Content-Type: 'application/yang-data+json'
          Accept: 'application/yang-data+json'
        body_format: json
        body:
        validate_certs: false
        status_code:
          - 200
          - 204
          - 404
      register: result

    - name: "View result"
      ansible.builtin.debug:
        msg: "{{ result['json']['Cisco-IOS-XE-crypto-pki-oper:crypto-pki-bundle'] }}"
      when: result.status == 200
```


Ansible - Using RESTCONF - Example output

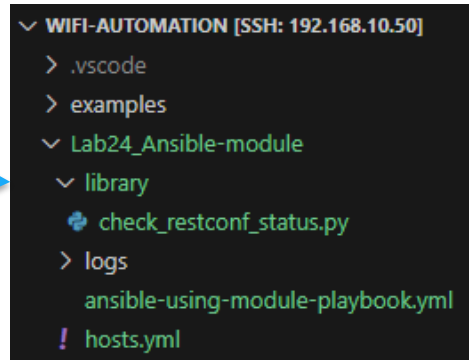
- Example output

```
(ansible-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab23_Ansible-RESTCONF$ ansible-playbook ansible-restconf-playbook.yml -i hosts.yml

PLAY [RESTCONF demo playbook] **********************************************************************************************************************

TASK [Get Certificate list with RESTCONF] **********************************************************************************************************
ok: [192.168.10.9]

TASK [View result] *********************************************************************************************************************************
ok: [192.168.10.9] => {
    "msg": [
        {
            "cert": [
                {
                    "subject-name": "cn=Cisco Licensing Root CA,o=Cisco",
                    "validity-end": "2038-05-30T19:48:47+00:00"
                }
            ],
            "label": "SLA-TrustPoint"
        },
        {
            "cert": [
                {
                    "subject-name": "cn=IOS-Self-Signed-Certificate-4022507931",
                    "validity-end": "2034-07-19T20:19:53+00:00"
                }
```

Ansible - Writing your own module - Intro

- If you want to do some stuff that is not supported in the builtin or community modules, you can write your own module. It can be small or large project, but it can also be something that is simple in Python, but that you would need to make complex structures in Ansible to solve
- The example will
  - Get "show run" from our device(s) using the cisco.ios.ios_facts module
  - Call our own module that use the input and return values based on that
  - Print the result
- This is a simple test/demo of a module. Some practical use of this module could be to check whether RESTCONF is enabled on a device, and based on the result of this test you could run other tasks as either RESTCONF, or fallback to CLI mode if RESTCONF is not available.

Ansible - Writing your own module - Tasks



- Task 0: Set up the files and folders and VS Code
  - Create a working directory and a playbook file "ansible-restconf-playbook.yml"
  - Copy the hosts.yml from previous projects
  - Create a "library" folder with a file "check_restconf_status.py" inside
  - Create a "logs" folder where we will dump some log files later
  - Remember to activate the automation-venv (use this for both Ansible and Python files in this exerc

Python 3.12.3 (ansible-venv)

```
devnet-adm@ubuntu-devnet:~/wifi-automation/Lab24_Ansible-module$ source ~/automation-venv/bin/activate
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab24_Ansible-module$ ansible --version
```

- Task 1: Create the Python module
  - Use the example in the Ansible docs "Developing modules" as an example https://docs.ansible.com/ansible/latest/dev_guide/developing_modules_general.html
- Task 2: Create the Ansible playbook
  - The playbook should have 3 tasks
    - Gather IOS facts (check docs on the cisco.ios.ios_facts module, https://docs.ansible.com/ansible/latest/collections/cisco/ios/ios_facts_module.html#ansible-collections-cisco-ios-ios-facts-module)
    - Run your module, it should be called the same way as a regular module, and the arguments indented below the module
    - Write the results to a YAML file

Ansible - Writing your own module - Hints for task 1

- I have removed all the docs from the example, you could probably keep it
- module args should be "hostname" and "show_run", both dict objects of type 'str'
- result should be dict, with changed=False and nothing else
- In the part where the actual "doing" is taking place, put stuff into result dict. You can add new key-value-pairs as you like
- Take the hostname from input parameters, put it in the result dict, under result['hostname']
- Check if "restconf" is in the variable module.params['show_run']
  - If yes, put "True" in result['restconf_enabled']
  - If no, put "False" in result['restconf_enabled']
- On exit, return the results back to Ansible

```python
# Written by Andreas Koksrud, based on framework from Erik Rongved and Espen Frøstrup.
from ansible.module_utils.basic import AnsibleModule

def run_module():
    # Define available arguments/parameters available to pass to the module from Ansible
    module_args = dict(
        hostname=dict(type='str', required=True),
        show_run=dict(type='str', required=True),
    )

    # Define the result object (which is passed back to Ansible)
    result = dict(
        changed=False
    )

    # Define the AnsibleModule object that will be instatiated when the module runs
    module = AnsibleModule(
        argument_spec=module_args,
        supports_check_mode=True
    )

    # From this point and down, is where the actual "doing" of the module takes place

    # We start populating the result object with the hostname of the WLC.
    result['hostname']=module.params['hostname']

    # Then we check if the text "restconf" is present in the run-config, and write some
    # corresponding values to the result object to pass back to Ansible
    if "restconf" in module.params['show_run']:
        result['restconf_enabled']=True
        result['comments']="RESTCONF status: Enabled"
    else:
        result['restconf_enabled']=False
        result['comments']="RESTCONF status: Disabled"

    # After doing the work, return the result back to Ansible as JSON
    module.exit_json(**result)


def main():
    run_module()

if __name__ == '__main__':
    main()
```


Ansible - Writing your own module - Hints for task 2

- Same playbook opening as we have used in the previous examples
- Gather IOS facts from the devices
- Call your module as you would any other module. As long as the name matches (without the .py extension) and it is located in the library folder, it will be found automatically
- Indented under the module you put the input variables
- Write the results to a YAML file
- Try to finish the line yourself (or look at next page)
- (optional) Try writing the results to JSON, it should be pretty similar, just format the content using "to_nice_json" instead

Lab24_Ansible-module > ansible-using-module-playbook.yml

```yaml
---

- name: Module demo
  hosts: wlc
  connection: network_cli
  gather_facts: false

  tasks:

    - name: Gather IOS facts
      cisco.ios.ios_facts:
        gather_subset: all
      register: facts

    - name: Check if RESTCONF is enabled
      check_restconf_status:
        hostname: "{{ facts.ansible_facts.ansible_net_hostname }}"
        show_run: "{{ facts.ansible_facts.ansible_net_config }}"
      register: result

    - name: Write results to YAML file
      ansible.builtin.copy:
        dest: "./logs/audit_{{ facts.ansible_facts.ansible_net_host
        mode: "0600"
        content: "{{ result | to_nice_yaml }}"
```

Ansible - Writing your own module - Example solution

check_restconf_status.py

```
# Written by Andreas Koksrud, based on framework from Erik Rongved and Espen Frøstrup.
from ansible.module_utils.basic import AnsibleModule

def run_module():
    # Define available arguments/parameters available to pass to the module from Ansible
    module_args = dict(
        hostname=dict(type='str', required=True),
        show_run=dict(type='str', required=True),
    )

    # Define the result object (which is passed back to Ansible)
    result = dict(
        changed=False
    )

    # Define the AnsibleModule object that will be instatiated when the module runs
    module = AnsibleModule(
        argument_spec=module_args,
        supports_check_mode=True
    )

    # From this point and down, is where the actual "doing" of the module takes place

    # We start populating the result object with the hostname of the WLC.
    result['hostname']=module.params['hostname']

    # Then we check if the text "restconf" is present in the run-config, and write some
    # corresponding values to the result object to pass back to Ansible
    if "restconf" in module.params['show_run']:
        result['restconf_enabled']=True
        result['comments']="RESTCONF status: Enabled"
    else:
        result['restconf_enabled']=False
        result['comments']="RESTCONF status: Disabled"

    # After doing the work, return the result back to Ansible as JSON
    module.exit_json(**result)


def main():
    run_module()

if __name__ == '__main__':
    main()
```

ansible-using-module-playbook.yml

```
---

- name: Module demo
  hosts: wlc
  connection: network_cli
  gather_facts: false

  tasks:

    - name: Gather IOS facts
      cisco.ios.ios_facts:
        gather_subset: all
      register: facts

    - name: Check if RESTCONF is enabled
      check_restconf_status:
        hostname: "{{ facts.ansible_facts.ansible_net_hostname }}"
        show_run: "{{ facts.ansible_facts.ansible_net_config }}"
      register: result

    - name: Write results to YAML file
      ansible.builtin.copy:
        dest: "./logs/audit_{{ facts.ansible_facts.ansible_net_hostname }} ({{ '%Y-%m-%d' | strftime }}).yml"
        mode: "0600"
        content: "{{ result | to_nice_yaml }}"
```

hosts.yml

```
wlc:
  hosts:
    192.168.10.9:
    192.168.10.30:
  vars:
    ansible_connection: network_cli
    ansible_network_os: ios
    ansible_ssh_pass: ChangeMe2025!
    ansible_password: ChangeMe2025!
    ansible_user: devnet-adm
    ansible_host_key_checking: False
```


Ansible - Writing your own module - Example output

- Example output

```
(ansible-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab24_Ansible-module$ ansible-playbook -i hosts.yml ansible-using-module-playbook.yml

PLAY [Module demo] ************************************************************************************************************************************************

TASK [Gather IOS facts] *******************************************************************************************************************************************
ok: [192.168.10.9]
ok: [192.168.10.30]

TASK [Check if RESTCONF is enabled] *******************************************************************************************************************************
ok: [192.168.10.30]
ok: [192.168.10.9]

TASK [Write results to YAML file] *********************************************************************************************************************************
changed: [192.168.10.30]
changed: [192.168.10.9]

PLAY RECAP ********************************************************************************************************************************************************
192.168.10.30              : ok=3    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
192.168.10.9               : ok=3    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

- Example of the written logfile (YAML)

Lab24_Ansible-module > logs > ! audit_wlc9 (2024-10-09).yml

```yaml
changed: false
comments: 'RESTCONF status: Enabled'
failed: false
hostname: wlc9
restconf_enabled: true
```

…. and the same, as JSON

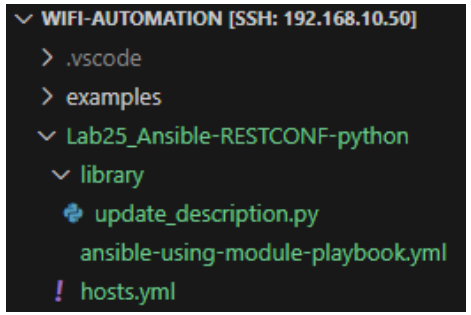
Lab24_Ansible-module > logs > {} audit_wlc9 (2024-10-09).json > ...

```json
{
    "changed": false,
    "comments": "RESTCONF status: Enabled",
    "failed": false,
    "hostname": "wlc9",
    "restconf_enabled": true
}
```


Ansible – Using RESTCONF + Python-module - Intro

- This is another exercise where we create our own Python module to do some "magic" with output from Ansible, and return them back to Ansible after being processed
- We will combine this with using RESTCONF directly from Ansible, as this is very fast compared to getting the same results using the command line
- By "very fast" that refers to the run-time, but there will often be some more investigation to do first to get the correct YANG module path, compared to your well-known CLI command to do the same
- The example will
  - Get all interfaces from the IOS-XE device using RESTCONF, including the descriptions
  - If the description of GigabitEthernet1 contains the word "Uplink" it will do nothing
  - If the description of GigabitEthernet1 does NOT contain the word "Uplink", it will create a new text for the description, with "Uplink" and a timestamp for the change
  - Push the updated interface config (description) back to the device using RESTCONF
- Be aware that this example takes a lot of shortcuts, just to show the concept. Like being valid only for GigabitEthernet1's description. And no error checking (try to run it if you have no description at all on the Gig1 interface…)

Ansible – Using RESTCONF + Python-module - Tasks



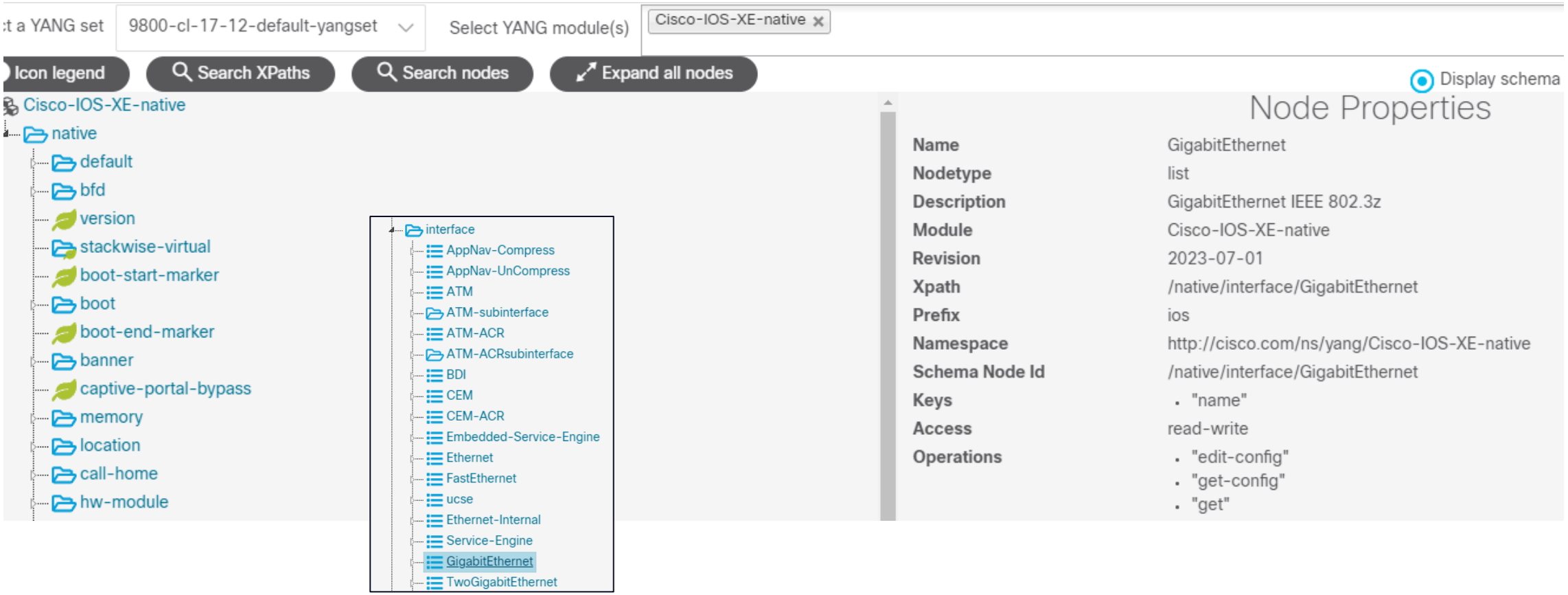
- Task 0: Set up the files and folders and VS Code
  - Create a working directory and a playbook file "ansible-using-module-playbook.yml"
  - Copy the hosts.yml from previous projects
  - Create a "library" folder with a file "update_description.py" inside
  - Remember to activate the automation-venv (use this for both Ansible and Python files in this exercise)

```
devnet-adm@ubuntu-devnet:~/wifi-automation/Lab25_Ansible-RESTCONF-python$ source ~/automation-
venv/bin/activate
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab25_Ansible-RESTCONF-python$ ansible --version
```

- Task 1: Find the correct RESTCONF path
  - Use YANG Suite or yangcatalog.org, check the module Cisco-IOS-XE-native and the :native/interface path
- Task 2: Create the Ansible playbook
  - Get interface list with RESTCONF (see Lab 23, but check out the YANG path Cisco-IOS-XE-native:native/interface instead)
  - Do something with the interface dictionary in a Python module (see Lab 24, but pass in restconf_result['json'] instead), return the interface dict to Ansible
  - Update the interface list/description using RESTCONF with the returned interface dict
  - 3 "extra" tasks could be "view result" tasks between each of your main tasks, for easier debugging
- Task 3: Create the Python module
  - Use the example in the Ansible docs "Developing modules", or the module from the previous lab as an example https://docs.ansible.com/ansible/latest/dev_guide/developing_modules_general.html

Ansible – Using RESTCONF + Python-module - Hints for Task 1

- Example on using YANG Suite to find the correct path



Ansible – Using RESTCONF + Python-module - Hints for Task 1

- Check in Postman:

```
GET  https://{{host}}/restconf/data/Cisco-IOS-XE-native:native/interface
```


```
{
    "Cisco-IOS-XE-native:interface": {
        "GigabitEthernet": [
            {
                "name": "1",
                "description": "Uplink",
                "switchport-config": {
                    "switchport": {
                        "Cisco-IOS-XE-switch:mode": {
                            "trunk": {}
                        },
                        "Cisco-IOS-XE-switch:trunk": {
                            "allowed": {
                                "vlan-v2": {
                                    "vlan-choices": {
                                        "vlans": "10"
                                    }
                                }
                            },
                            "native": {
```



```
{
    "Cisco-IOS-XE-native:interface": {
        "GigabitEthernet": [
            {
                "name": "1",
                "description": "Uplink"
            },
            {
                "name": "2",
                "description": "Out-of-band"
            },
            {
                "name": "3"
            }
        ]
    }
}
```


- Then we choose only a couple of fields, and try again

```
GET  https://{{host}}/restconf/data/Cisco-IOS-XE-native:native/interface?fields=GigabitEthernet/name;GigabitEthernet/description
```

Ansible – Using RESTCONF + Python-module - Hints for Task 2

- We start by creating the task that gets the data from WLC, using the RESTCONF path we have tested in Postman

```yaml
---

- name: Interface Playbook
  hosts: wlc
  connection: network_cli
  gather_facts: false

  tasks:

    - name: Get Interface list with RESTCONF
      ansible.builtin.uri:
        url: "https://{{ ansible_host }}/restconf/data/Cisco-IOS-XE-native:native/interface?fields=GigabitEthernet/name;GigabitEthernet/description"
        user: "{{ ansible_user }}"
        password: "{{ ansible_password }}"
        method: GET
        headers:
          Content-Type: 'application/yang-data+json'
          Accept: 'application/yang-data+json'
        body_format: json
        body:
        validate_certs: false
        status_code:
          - 200
          - 204
          - 404
      register: restconf_result
```

- We create a simple "view" task, to easier understand and debug what we have got out. Only if the status code = 200.

```yaml
    - name: View result from RESTCONF
      ansible.builtin.debug:
        msg: "{{ restconf_result }}"
      when: restconf_result.status == 200
```


Ansible – Using RESTCONF + Python-module - Hints for Task 2

- Then we create a task that will send the interface dictionary (in JSON format) to our Python module for processing
- We only do this if (when:) the status code from our RESTCONF call equals 200 ("OK")

```
33      - name: Do something with the interface list in a Python module
34        update_description:
35          interface_dict: "{{ restconf_result['json'] }}"
36        register: update_description_result
37        when: restconf_result.status == 200
```

- Again, we create a simple "view" task to view the results/output of the Python module processing, which we have registered in the variable "update_description_result"
- And again, only if the result of the original RESTCONF call was "200 OK"

```
39      - name: View result
40        ansible.builtin.debug:
41          msg: "{{ update_description_result }}"
42        when: restconf_result.status == 200
```


Ansible – Using RESTCONF + Python-module - Hints for Task 2

- The last part of the playbook, you might want to write after or in parallel with writing the Python module as it includes using the output from the module
- We create the final of our three primary tasks for this playbook, updating the WLC interface description using RESTCONF
- Notice that the use of the uri module is similar to the read task, only using PATCH instead of GET, and actually having some data in the "body" field. What we put in there is based on what we got out, only modified in the Python module which we will look into next

```
44      - name: Update Interface list with RESTCONF
45        ansible.builtin.uri:
46          url: "https://{{ ansible_host }}/restconf/data/Cisco-IOS-XE-native:native/interface"
47          user: "{{ ansible_user }}"
48          password: "{{ ansible_password }}"
49          method: PATCH
50          headers:
51            Content-Type: 'application/yang-data+json'
52          body_format: json
53          body: "{{ update_description_result.new_interface_dict }}"
54          validate_certs: false
55          status_code:
56            - 200
57            - 204
58            - 404
59        register: restconf_write_result
```

- As usual, we have a "view" task to check the output

```
61      - name: View result from RESTCONF write
62        ansible.builtin.debug:
63          msg: "{{ restconf_write_result }}"
64        when: restconf_write_result.status == 204
```


Ansible – Using RESTCONF + Python-module - Hints for Task 3

- Now we are ready to write the Python module "update_description.py". Lots of it will be identical to the previous Lab, but we have only one input parameter now. This will be the dictionary with the output from the RESTCONF call.

```
5   def run_module():
6       # Define available arguments/parameters available to pass to the module from Ansible
7       module_args = dict(
8           interface_dict=dict(type='dict', required=True),
9       )
```

- Then we do the actual reading and manipulation of the input

```
22      # From this point and down, is where the actual "doing" of the module takes place
23
24      interface_dict = module.params['interface_dict']
25      timestamp = datetime.now().strftime('%D %H:%M:%S')
26      description = interface_dict['Cisco-IOS-XE-native:interface']['GigabitEthernet'][0]['description']
27
28      # If the description field contains "Uplink" we will do nothing. If not, we will change the description
29      if "Uplink" in description:
30          result['Gi1_old_description'] = description
31          result['Gi1_new_description'] = description
32          result['new_interface_dict'] = interface_dict
33          result['comments'] = f"Unchanged"
34      else:
35          result['Gi1_old_description'] = description
36          result['Gi1_new_description'] = f"Uplink - modified {timestamp}"
37          interface_dict['Cisco-IOS-XE-native:interface']['GigabitEthernet'][0]['description'] = f"Uplink - mo
38          result['new_interface_dict'] = interface_dict
39          result['changed'] = True
40          result['comments'] = f"Changed {timestamp}"
41
42      # After doing the work, return the result back to Ansible as JSON
43      module.exit_json(**result)
```

Put the input dictionary in a variable

Extract the description text. An intermediate step here could be to do a print() of the interface_dict

Check if the text "Uplink" is in the description. If it is, we do nothing

If there is no "Uplink" text in the description, we create a new description text, including a timestamp

Then, we put the updated interface dictionary into the "result" dictionary, so Ansible can use it. Note here, that we put it under the key ['new_interface_dict'] which is where we extract it from in the Ansible playbook (line 53)

```
53          body: "{{ update_description_result.new_interface_dict }}"
```


Ansible – Using RESTCONF + Python-module - Hints for Task 3

- You can test the writing part in Postman

```
9800-vm#show run int g1
Building configuration...

Current configuration : 192 bytes
!
interface GigabitEthernet1
 description Uplink
```

- Change "GET" to "PATCH"

PATCH https://{{host}}/restconf/data/Cisco-IOS-XE-native:native/interface

Params Authorization Headers (12) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
{
    "Cisco-IOS-XE-native:interface": {
        "GigabitEthernet": [
            {
                "name": "1",
                "description": "Test12341234"
            }
        ]
    }
}
```

- Put the modified body field in the "body" section

Send

Status: 204 No Content Time: 256 ms Size: 499 B

```
9800-vm#show run int g1
Building configuration...

Current configuration : 198 bytes
!
interface GigabitEthernet1
 description Test12341234
 switchport trunk native vlan 10
 switchport trunk allowed vlan 10
```


Ansible – Using RESTCONF + Python-module - Hints for Task 3

- This is the same part we tested in Postman, only in Ansible

```yaml
ansible.builtin.uri:
  url: "https://{{ ansible_host }}/restconf/data/Cisco-IOS-XE-native:native/interface"
  user: "{{ ansible_user }}"
  password: "{{ ansible_password }}"
  method: PATCH
  headers:
    Content-Type: 'application/yang-data+json'
  body_format: json
  body: "{{ result_c.test_result.new_interface_dict }}"
  validate_certs: false
  status_code:
    - 200
    - 204
    - 404
register: result
```

```
9800-vm#show run int g1
Building configuration...

Current configuration : 198 bytes
!
interface GigabitEthernet1
 description Test12341234
 switchport trunk native vlan 10
 switchport trunk allowed vlan 10
```

Using the variable that was updated with the Python module

```
/Lab25_Ansible-RESTCONF-python$ ansible-playbook -i hosts.yml ansible-using-module-playbook.yml

TASK [Test C: Update Interface list with RESTCONF] ***************
ok: [192.168.10.10]
ok: [192.168.10.9]
```

```
9800-vm#show run int g1
Building configuration...

Current configuration : 221 bytes
!
interface GigabitEthernet1
 description Uplink - modified 01/29/24 12:55:54
 switchport trunk native vlan 10
 switchport trunk allowed vlan 10
 switchport mode trunk
```


Ansible – Using RESTCONF + Python-module - Example solution

interface-playbook.yaml

```
---

- name: Interface Playbook
  hosts: wlc
  connection: network_cli
  gather_facts: false

  tasks:

    - name: Get Interface list with RESTCONF
      ansible.builtin.uri:
        url: "https://{{ ansible_host }}/restconf/data/Cisco-IOS-XE-
native:native/interface?fields=GigabitEthernet/name;GigabitEthernet/description"
        user: "{{ ansible_user }}"
        password: "{{ ansible_password }}"
        method: GET
        headers:
          Content-Type: 'application/yang-data+json'
          Accept: 'application/yang-data+json'
        body_format: json
        body:
        validate_certs: false
        status_code:
          - 200
          - 204
          - 404
      register: restconf_result

    - name: Do something with the interface list in a Python module
      update_description:
        interface_dict: "{{ restconf_result['json'] }}"
      register: update_description_result
      when: restconf_result.status == 200

    - name: View result
      ansible.builtin.debug:
        msg: "{{ update_description_result }}"
      when: restconf_result.status == 200

    - name: Update Interface list with RESTCONF
      ansible.builtin.uri:
        url: "https://{{ ansible_host }}/restconf/data/Cisco-IOS-XE-native:native/interface"
        user: "{{ ansible_user }}"
        password: "{{ ansible_password }}"
        method: PATCH
        headers:
          Content-Type: 'application/yang-data+json'
        body_format: json
        body: "{{ update_description_result.new_interface_dict }}"
        validate_certs: false
        status_code:
          - 200
          - 204
          - 404
      register: restconf_write_result

    - name: View result from RESTCONF write
      ansible.builtin.debug:
        msg: "{{ restconf_write_result }}"
      when: restconf_write_result.status == 204
```

update_description.py

```
# Written by Andreas Koksrud, based on framework from Erik Rongved and Espen Frøstrup.
from ansible.module_utils.basic import AnsibleModule
from datetime import datetime

def run_module():
    # Define available arguments/parameters available to pass to the module from Ansible
    module_args = dict(
        interface_dict=dict(type='dict', required=True),
    )

    # Define the result object (which is passed back to Ansible)
    result = dict(
        changed=False
    )

    # Define the AnsibleModule object that will be instatiated when the module runs
    module = AnsibleModule(
        argument_spec=module_args,
        supports_check_mode=True
    )

    # From this point and down, is where the actual "doing" of the module takes place

    interface_dict = module.params['interface_dict']
    timestamp = datetime.now().strftime('%D %H:%M:%S')
    description = interface_dict['Cisco-IOS-XE-native:interface']['GigabitEthernet'][0]['description']

    # If the description field contains "Uplink" we will do nothing. If not, we will change the description including
a timestamp just for fun
    if "Uplink" in description:
        result['Gi1_old_description'] = description
        result['Gi1_new_description'] = description
        result['new_interface_dict'] = interface_dict
        result['comments'] = f"Unchanged"
    else:
        result['Gi1_old_description'] = description
        result['Gi1_new_description'] = f"Uplink - modified {timestamp}"
        interface_dict['Cisco-IOS-XE-native:interface']['GigabitEthernet'][0]['description'] = f"Uplink - modified
{timestamp}"
        result['new_interface_dict'] = interface_dict
        result['changed'] = True
        result['comments'] = f"Changed {timestamp}"

    # After doing the work, return the result back to Ansible as JSON
    module.exit_json(**result)


def main():
    run_module()

if __name__ == '__main__':
    main()
```

hosts.yml

```
wlc:
  hosts:
    192.168.10.9:
    192.168.10.30:
  vars:
    ansible_connection: network_cli
    ansible_network_os: ios
    ansible_ssh_pass: ChangeMe2025!
    ansible_password: ChangeMe2025!
    ansible_user: devnet-adm
    ansible_host_key_checking: False
```

WLC before changes

```
wlc9#show run int g1
Building configuration...

Current configuration : 164 bytes
!
interface GigabitEthernet1
 description Test123
 switchport trunk native vlan 10
 switchport trunk allowed vlan 10,11
 switchport mode trunk
 negotiation auto
end
```

WLC after changes

```
wlc9#show run int g1
Building configuration...

Current configuration : 194 bytes
!
interface GigabitEthernet1
 description Uplink - modified 10/15/24 20:25:40
 switchport trunk native vlan 10
 switchport trunk allowed vlan 10,11
 switchport mode trunk
 negotiation auto
end
```


Ansible – Using RESTCONF + Python-module - Example output

- One of the WLCs were not available
- The Python module did something to the input. Since we put result['changed']=True, it will report that the task did change something
- View result from the Python module, here we can see the full "result", including the new interface dict which we use in the RESTCONF task to write to the WLC next
- Here we have written to WLC using RESTCONF
- And print the write result

```
(ansible-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab25_Ansible-RESTCONF-python$ ansible-playbook -i hosts.yml ansible-using-module-playbook.yml

PLAY [Interface Playbook] ******************************************************************************************************************************************************************

TASK [Get Interface list with RESTCONF] ****************************************************************************************************************************************************
ok: [192.168.10.9]
fatal: [192.168.10.30]: FAILED! => {"changed": false, "elapsed": 3, "msg": "Status code was -1 and not [200, 204, 404]: Request failed: <urlopen error [E
e, "status": -1, "url": "https://192.168.10.30/restconf/data/Cisco-IOS-XE-native:native/interface?fields=GigabitEthernet/name;GigabitEthernet/description"

TASK [Do something with the interface list in a Python module] *****************************************************************************************************************************
changed: [192.168.10.9]

TASK [View result] *************************************************************************************************************************************************************************
ok: [192.168.10.9] => {
    "msg": {
        "Gi1_new_description": "Uplink - modified 10/15/24 20:25:40",
        "Gi1_old_description": "jalla",
        "changed": true,
        "comments": "Changed 10/15/24 20:25:40",
        "failed": false,
        "new_interface_dict": {
            "Cisco-IOS-XE-native:interface": {
                "GigabitEthernet": [
                    {
                        "description": "Uplink - modified 10/15/24 20:25:40",
                        "name": "1"
                    }
                ]
            }
        }
    }
}

TASK [Update Interface list with RESTCONF] *************************************************************************************************************************************************
ok: [192.168.10.9]

TASK [View result from RESTCONF write] *****************************************************************************************************************************************************
ok: [192.168.10.9] => {
    "msg": {
        "cache_control": "private, no-cache, must-revalidate, proxy-revalidate",
        "changed": false,
        "connection": "close",
        "content_security_policy": "default-src 'self'; block-all-mixed-content; base-uri 'self'; frame-ancestors 'none';, default-src 'self'; script-src
        "content_type": "text/html; charset=UTF-8",
        "cookies": {}
```

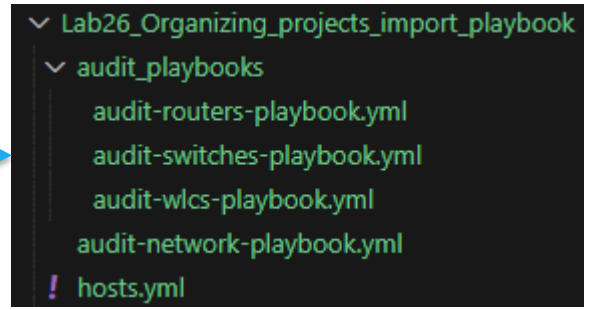

Organizing projects using import_playbook - Intro

- For larger projects, there are several ways to organize stuff
- I like to break projects down into as small parts as possible
  - Each part will be easy to both maintain and understand
  - Code will be easier to reuse, change and troubleshoot
- A playbook can run other playbooks by using the import_playbook module, which we will do in this exercise
- A playbook can also call tasks by using the roles module, which is what we will do in the next exercise
- General best practices (for Ansible and everything else…)
  - Try to keep each part as simple as possible
  - Use whitespace and comments to break things up and explain generously. What is obvious to you will probably not be obvious to everyone
- In the following task we will create a (somewhat constructed) example using some of these techniques

- Reference: https://docs.ansible.com/ansible/latest/tips_tricks/sample_setup.html#sample-directory-layout

Organizing projects using import_playbook - Tasks

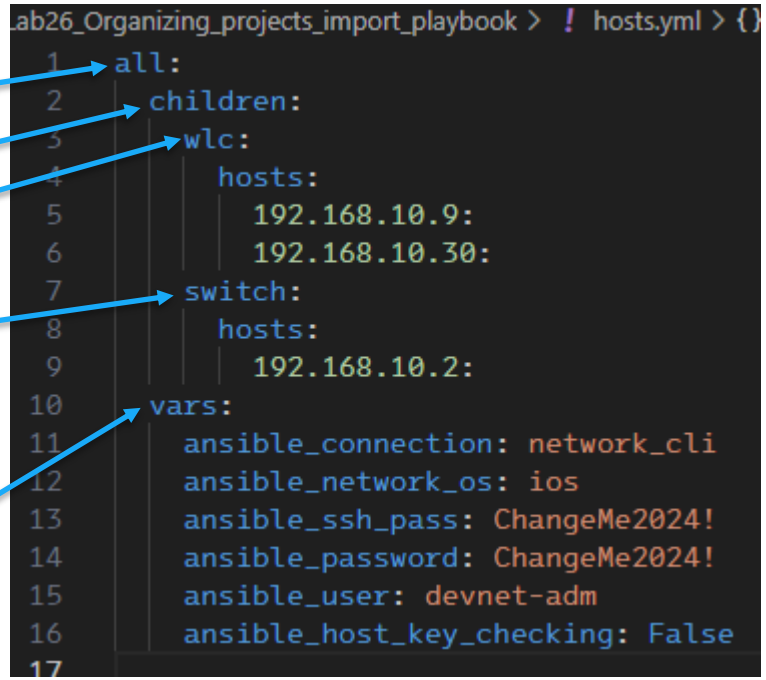
- Task 0: Set up the files and folders and VS Code
  - Create directories according to this example
  - Create playbook files according to this example



- Task 1: Create an extended variant of our hosts.yml file
  - The inventory should include
    - Shared switch (192.168.10.2)
    - Your WLC (192.168.10.{YOUR_WLC_IP})
    - Shared WLC (192.168.10.9)
- Task 2: Create the three playbooks in the "audit_playbooks" folder
  - They should be simple playbooks, use what you have learned. The example use two tasks:
    - Gather IOS facts (using cisco.ios.ios_facts module)
    - Output hostname and version (using ansible.builtin.debug module and prints text from the facts)
- Task 3: Create the audit-network-playbook.yml playbook which is used to run the other three playbooks using the ansible.builtin.import_playbook module

Organizing projects using import_playbook - Hints for Task 1

- To accommodate for more than only WLCs, we create a slightly more advanced structure for our inventory file than we have done until now
- Top-level group is "all"
- Under "all" we have "children"
- Under the children we have "wlc" and "switch"



- The "vars" are common for all devices, so we have it directly under the "all" group, on the same level as "children"
- It is also possible to have "vars" under each child group (on the same level as "hosts"). This is not used here.
- It is also possible to have vars under each host. This is not used here.

Organizing projects using import_playbook - Hints for Task 2

- This is the "audit-wlcs-playbook.yml". The others in the example will be identical except the task name
- For real-world the tasks for each device type would probably differ, else they don't need to be separate ☺
- We start with the usual playbook opening we have used in the previous tasks

```yaml
---
- name: Audit WLCs
  hosts: wlc
  connection: network_cli
  gather_facts: false
```

- Then we have a "Gather IOS facts" task using the "cisco.ios.ios_facts" module: Register the result in a variable for later use

```yaml
  tasks:
    - name: Gather IOS facts
      cisco.ios.ios_facts:
        gather_subset: all
      register: facts1
```

- Lastly we have a output task, using "ansible.builtin.debug" module to output parts of the registered results.
- Notice we use Jinja2 here. To see the full contents of "facts1" you could run this using only `msg: {{ facts1 }}` to see what is contained within the results. Then expand to `msg: {{ facts1.ansible_facts }}` before using the single values in your text

```yaml
    - name: Output hostname and version
      ansible.builtin.debug:
        msg: "Hostname: {{ facts1.ansible_facts.ansible_net_hostname }} (version {{ facts1.ansible_facts.ansible_net_version }})"
```

- Copy-paste this playbook to similar variants for router and switch (change hosts: wlc to hosts: switch and hosts: router)

Organizing projects using import_playbook - Hints for Task 3

- In the main playbook "audit-network-playbook.yml" we have 3 tasks with only one line in each. They simply run the other playbooks (from the subfolder)

Lab26_Organizing_projects_import_playbook > audit-network-playbook.yml

```yaml
---
- name: Audit WLCs
  ansible.builtin.import_playbook: audit_playbooks/audit-wlcs-playbook.yml
- name: Audit switches
  ansible.builtin.import_playbook: audit_playbooks/audit-switches-playbook.yml
- name: Audit routers
  ansible.builtin.import_playbook: audit_playbooks/audit-routers-playbook.yml
```


Organizing projects using import_playbook - Example solution

audit-network-playbook.yml

```
---
- name: Audit WLCs
  ansible.builtin.import_playbook: audit_playbooks/audit-wlcs-playbook.yml
- name: Audit switches
  ansible.builtin.import_playbook: audit_playbooks/audit-switches-playbook.yml
- name: Audit routers
  ansible.builtin.import_playbook: audit_playbooks/audit-routers-playbook.yml
```

hosts.yml

```
all:
  children:
    wlc:
      hosts:
        192.168.10.9:
        192.168.10.30:
    switch:
      hosts:
        192.168.10.2:
  vars:
    ansible_connection: network_cli
    ansible_network_os: ios
    ansible_ssh_pass: ChangeMe2025!
    ansible_password: ChangeMe2025!
    ansible_user: devnet-adm
    ansible_host_key_checking: False
```

audit-routers-playbook.yml

```
---
- name: Audit routers
  hosts: router
  connection: network_cli
  gather_facts: false
  tasks:
    - name: Gather IOS facts
      cisco.ios.ios_facts:
        gather_subset: all
      register: facts1
    - name: Output hostname and version
      ansible.builtin.debug:
        msg: "Hostname: {{ facts1.ansible_facts.ansible_net_hostname }} (version {{ facts1.ansible_facts.ansible_net_version }})"
```

audit-switches-playbook.yml

```
---
- name: Audit switches
  hosts: switch
  connection: network_cli
  gather_facts: false
  tasks:
    - name: Gather IOS facts
      cisco.ios.ios_facts:
        gather_subset: all
      register: facts1
    - name: Output hostname and version
      ansible.builtin.debug:
        msg: "Hostname: {{ facts1.ansible_facts.ansible_net_hostname }} (version {{ facts1.ansible_facts.ansible_net_version }})"
```

audit-wlcs-playbook.yml

```
---
- name: Audit WLCs
  hosts: wlc
  connection: network_cli
  gather_facts: false
  tasks:
    - name: Gather IOS facts
      cisco.ios.ios_facts:
        gather_subset: all
      register: facts1
    - name: Output hostname and version
      ansible.builtin.debug:
        msg: "Hostname: {{ facts1.ansible_facts.ansible_net_hostname }} (version {{ facts1.ansible_facts.ansible_net_version }})"
```


Organizing projects using import_playbook - Example output

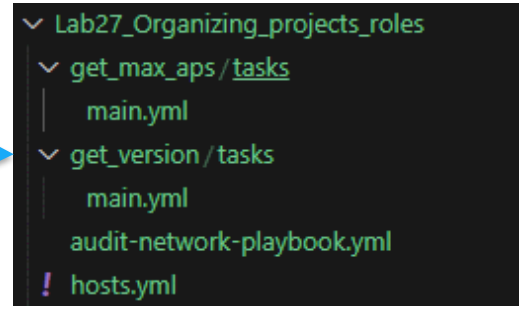
- Since we do not have a "router" group in our hosts.yml file, we get a warning when we run the playbook since it does not match any host groups

Organizing projects using Roles - Intro

- Instead of using a master playbook to run other playbooks, in this exercise we will use "roles" to call tasks on each device that has that role
- The example will still be somewhat constructed and simplified to primarily show the concept of roles rather than being super-useful in the real world by itself
- This example contains two roles
  - get_version: This role gets and prints the current version of the device, and is applied to all our devices (reuse some of the playbook from Lab 5)
  - get_max_aps: This gets the maximum number of APs on a WLC, it is applied only to the WLC (reuse playbook from Lab 20)

- Reference: https://docs.ansible.com/ansible/latest/network/getting_started/network_roles.html

Organizing projects using Roles - Tasks



- Task 0: Set up the files and folders and VS Code
  - Create directory `get_max_aps` and subdirectory `tasks`, wherein you create `main.yml`
  - Create directory `get_version` and subdirectory `tasks`, wherein you create `main.yml`
  - Create the `audit-network-playbook.yml`
  - Copy the `hosts.yml` file from the previous exercise
- Task 1: Create the audit-network-playbook.yml content
  - Playbook task 1: Get version from all Cisco devices, for "all" hosts, using the roles "get_version"
  - Playbook task 2: Get max APs from WLCs, for "wlc" hosts, using the roles "get_max_aps"
- Task 2: Create the role get_version
  - Edit the main.yml in the get_version/tasks folder. It should have two tasks:
    - Gather IOS facts. Reuse from Lab 5, but only the task, not the playbook stuff before the task. You will need to de-indent this to avoid indentation errors since you drop the "outer layer" in the yaml file
    - Ouput task, use the ansible.builtin.debug module instead of writing to file as you did in Lab 5. You can narrow down the output to only ansible_net_hostname and ansible_net_version with some enclosing text
- Task 3: Create role get_max_aps
  - Edit the main.yml in the get_max_aps/tasks folder. It should have two tasks:
    - Gather Max APs supported. Reuse from Lab 20, but only the task, not the playbook stuff before the task. You will need to de-indent this to avoid indentation errors since you drop the "outer layer" in the yaml file
    - Ouput task, use the ansible.builtin.debug module, you can print the {{ result.stdout }} message

Organizing projects using Roles - Hints for Task 1

- We start with creating the audit-network-playbook.yml content
- It consists of two tasks, as you can see there is not any explicit "doing" in these tasks, they just have some roles assigned
- To expand this example, you could have more roles under the relevant tasks for the different device groups. For instance, the roles under the first task could also include roles "get_restconf_status", "backup_run_config" etc. The WLC specific task could have roles for "get_ap_summary", "get_wireless_clients" etc
- Our reusable roles are entered under the "roles" section in each task

Lab27_Organizing_projects_roles > audit-network-playbook.yml

```
1  ---
2  - name: Get version from all Cisco devices
3    hosts: all
4    connection: network_cli
5    gather_facts: false
6    roles:
7      - get_version
8
9  - name: Get max APs
10   hosts: wlc
11   connection: network_cli
12   gather_facts: false
13   roles:
14     - get_max_aps
15
```


Organizing projects using Roles - Hints for Task 2

- To "create" the role get_version we have created the subfolder "get_version" with another subfolder "tasks", in where there is a file named "main.yml". This structure is identical for all roles, just change the upper folder name
- As you can see from the example below, the role itself does just contain the same tasks that we create in Lab5, except that we output the results to the terminal using debug, instead of writing to a file

Lab27_Organizing_projects_roles > get_version > tasks > main.yml

```yaml
---
- name: Gather IOS facts
  cisco.ios.ios_facts:
    gather_subset: all
  register: facts1

- name: Output hostname and version
  ansible.builtin.debug:
    msg: "Hostname: {{ facts1.ansible_facts.ansible_net_hostname }} (version {{ facts1.ansible_facts.ansible_net_version }})"

```


Organizing projects using Roles - Hints for Task 3

- The "get_max_aps" role is created in the same way as the previous role. You just have to have the folder named after your role, create a "tasks" folder inside that, and the "main.yml" file in the tasks folder
- The tasks in the role main.yml are copy-paste from Lab20, just remember to de-indent them one level, since they are on their own and not under a "tasks:" statement

Lab27_Organizing_projects_roles > get_max_aps > tasks > main.yml

```yaml
---
- name: "Get Max APs supported with CLI command"
  cisco.ios.ios_command:
    commands: "show wireless summary | include Max APs"
  register: result

- name: "View result"
  ansible.builtin.debug:
    msg: "{{ result.stdout }}"

```


Organizing projects using Roles - Example solution

audit-network-playbook.yml

```
---
- name: Get version from all Cisco devices
  hosts: all
  connection: network_cli
  gather_facts: false
  roles:
    - get_version

- name: Get max APs
  hosts: wlc
  connection: network_cli
  gather_facts: false
  roles:
    - get_max_aps
```

hosts.yml

```
all:
  children:
    wlc:
      hosts:
        192.168.10.9:
        192.168.10.30:
    switch:
      hosts:
        192.168.10.2:
  vars:
    ansible_connection: network_cli
    ansible_network_os: ios
    ansible_ssh_pass: ChangeMe2025!
    ansible_password: ChangeMe2025!
    ansible_user: devnet-adm
    ansible_host_key_checking: False
```

get_max_aps/tasks/main.yml

```
---
- name: "Get Max APs supported with CLI command"
  cisco.ios.ios_command:
    commands: "show wireless summary | include Max APs"
  register: result

- name: "View result"
  ansible.builtin.debug:
    msg: "{{ result.stdout }}"
```

get_version/tasks/main.yml

```
---
- name: Gather IOS facts
  cisco.ios.ios_facts:
    gather_subset: all
  register: facts1

- name: Output hostname and version
  ansible.builtin.debug:
    msg: "Hostname: {{ facts1.ansible_facts.ansible_net_hostname }} (version {{ facts1.ansible_facts.ansible_net_version }})"
```

```
(ansible-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab27_Organizing_projects_roles$ tree
.
├── audit-network-playbook.yml
├── get_max_aps
│   └── tasks
│       └── main.yml
├── get_version
│   └── tasks
│       └── main.yml
└── hosts.yml

5 directories, 4 files
```


Organizing projects using Roles - Example output

```
(ansible-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab27_Organizing_projects_roles$ ansible-playbook -i hosts.yml audit-network-playbook.yml

PLAY [Get version from all Cisco devices] ***************************************************************************************************************

TASK [get_version : Gather IOS facts] *******************************************************************************************************************
ok: [192.168.10.9]
ok: [192.168.10.30]
ok: [192.168.10.2]

TASK [get_version : Output hostname and version] ********************************************************************************************************
ok: [192.168.10.9] => {
    "msg": "Hostname: wlc9 (version 17.15.01)"
}
ok: [192.168.10.2] => {
    "msg": "Hostname: switch-2 (version 15.2(7)E10)"
}
ok: [192.168.10.30] => {
    "msg": "Hostname: wlc30 (version 17.15.01)"
}

PLAY [Get max APs] **************************************************************************************************************************************

TASK [get_max_aps : Get Max APs supported with CLI command] *********************************************************************************************
ok: [192.168.10.9]
ok: [192.168.10.30]

TASK [get_max_aps : View result] ************************************************************************************************************************
ok: [192.168.10.9] => {
    "msg": [
        "Max APs supported          : 100"
    ]
}
ok: [192.168.10.30] => {
    "msg": [
        "Max APs supported          : 100"
    ]
}

PLAY RECAP **********************************************************************************************************************************************
192.168.10.2               : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
192.168.10.30              : ok=4    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
192.168.10.9               : ok=4    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

(ansible-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab27_Organizing_projects_roles$
```


Next steps suggestions

- A possible next step after doing these labs, would be to look into AWX
  - Web based GUI, great for organizing and orchestrating
  - Task engine for scheduling playbooks etc

- Using NetBox for inventory management and single source of truth
  - Ansible playbooks can run based on an inventory in NetBox
  - Ansible playbooks can populate fields in NetBox
  - Fields (standard or custom) in NetBox can decide stuff in Ansible
    - E.g.: is it a WLC? -> Run this task

- Any ideas for new exercises to add to this deep dive? What would you like to see in the next version?

In-depth: Python

- This part of the Day 2 labs will focus on using Python
- Python for automation can be used directly, or as a building block in other tools/solutions

- !!! As you progress through the script and tasks, you will come across missing packages !!!
  - Install them in your VENV (automation-venv) with the following command (netmiko used as example)

```
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/ Lab40_Python_Netmiko$ pip install netmiko
```

  - If you are not in your venv, you enter using this command. It will be repeated in the Task 0 of each exercise.

```
devnet-adm@ubuntu-devnet:~/wifi-automation/Lab40_Python_Netmiko$ source ~/automation-venv/bin/activate
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/ Lab40_Python_Netmiko$
```

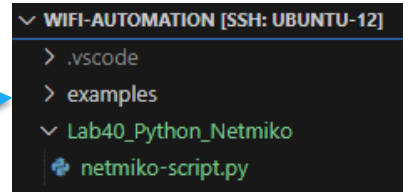
  - You will see which venv you are in by the (automation-venv) at the beginning of each line
  - To "get out" of your venv, you can run the "deactivate" command, or restart the terminal/SSH

```
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/ Lab40_Python_Netmiko$ deactivate
devnet-adm@ubuntu-devnet:~/wifi-automation/Lab40_Python_Netmiko$
```


Python - Using Netmiko - Intro

- In this exercise, we will write a python script that connects to the WLC using SSH
- The script will
  - Take the WLC IP as input
  - Do a user/pass prompt before connecting
  - Have a menu with these choices
    1. Show AP summary
    2. Show client summary
    3. Show CDP neighbors
    4. Save run-config to a timestamped file
- You can extend the script as you like, be creative
- There is not much error checking, you can put in stuff like checking if it is really a WLC, etc
- Reference: https://github.com/ktbyers/netmiko/blob/develop/README.md

Python - Using Netmiko - Tasks



- Task 0: Set up the files and folders and VS Code
  - Create a working directory
  - Create a Python file called "netmiko-script.py"
  - Activate the automation-venv 3.12.3 ('python-lab-venv': venv)

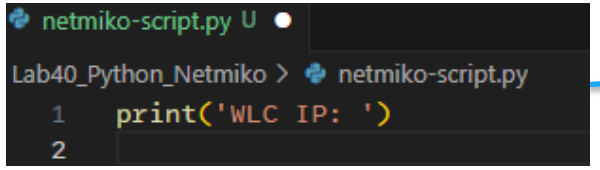
```
devnet-adm@ubuntu-devnet:~/wifi-automation/Lab40_Python_Netmiko$ source ~/automation-venv/bin/activate
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/ Lab40_Python_Netmiko$ python --version
```

- Task 1: Write a python script that
  - Take the WLC IP as input
  - Prints the WLC IP with some text as output, then quits
- Task 2: Extend the script, so it will
  - Ask for username and password
  - Connect to the IP using the username and password
  - Do a show command (show ip interfaces brief or something), then quits
- (optional) Task 3: Extend the script, so it will
  - Present a menu with the 4 choices from previous slide
  - Run the commands, then quit

Python - Using Netmiko - Hints: Task 1

- Task 1: Write a python script that
  - Take the WLC IP as input
  - Prints the WLC IP with some text as output, then quits

- Start with the print command, printing a static text




```
40_Python_Netmiko$ python netmiko-script.py
WLC IP:
```

- Then add the input, which is saved to a variable "wlc_ip"

```
1 wlc_ip = input('Enter WLC IP: ')
2 print(f'The WLC IP is: {wlc_ip}')
```

```
40_Python_Netmiko$ python netmiko-script.py
Enter WLC IP: 1.2.3.4
The WLC IP is: 1.2.3.4
```

- Notice the "f-string" formatting of the printed text. This is widely used in Python to use variables inside strings.

- To keep this short and simple, we have no error checking. So the "IP" you enter could just as well be "blablabla"

```
40_Python_Netmiko$ python netmiko-script.py
Enter WLC IP: blablabla
The WLC IP is: blablabla
```

- This will be the case in most of these labs, it is by design
- Depending on the usage of your script, you should probably add some error checking, especially if used by others. One example would be to use regex to check if the value is a correct IP address, so the output could be like this

```
40_Python_Netmiko$ python netmiko-script.py
Enter WLC IP: blablabla
Invalid IP address, exiting...
```


Python - Using Netmiko - Hints: Task 2

- Task 2: Extend the script, so it will
  - Ask for username and password
  - Connect to the IP using the username and password
  - Do a show command (show ip interfaces brief or something), then quits

- Building on the work from Task 1, we add user/pass input

```
1  import getpass
2  wlc_ip = input('WLC IP: ')
3  username = input('Username: ')
4  password = getpass.getpass('Password: ')
```

- Now, we will make Netmiko connect and run a command
  - import the ConnectHandler function from Netmiko

```
2  from netmiko import ConnectHandler
```

  - Create a "cisco_wlc" dictionary which ConnectHandler use as input

```
8  cisco_wlc = {
9      'device_type': 'cisco_ios',
10     'ip': wlc_ip,
11     'username': username,
12     'password': password
13 }
14 net_connect = ConnectHandler(**cisco_wlc)
```

  - Create a "command" variable, which we print, and send to the WLC

```
15 command = "show ip int brief"
16 print("Command: " + command)
17 print(net_connect.send_command(command))
18 net_connect.disconnect()
```

- Example output from running the script so far

```
40_Python_Netmiko$ python netmiko-script.py
Enter WLC IP: 192.168.10.9
Username: devnet-adm
Password:
Command: show ip int brief
Interface              IP-Address      OK? Method
GigabitEthernet1       unassigned      YES unset
GigabitEthernet2       unassigned      YES unset
GigabitEthernet3       unassigned      YES unset
Vlan1                  unassigned      YES NVRAM
Vlan10                 192.168.10.10   YES NVRAM
```


Python - Using Netmiko - Hints: Task 3

- To accomplish this you should separate the tasks into functions
- You write a function this way

```
def my_function():
    print("Hello, I am a function!")
```

- To use the function, use the function name

```
my_function()
```

- An example structure for Task3 can be:
  - imports (same as Task2)
  - Define functions for each menu choice (hints on next page)
  - Setup netmiko

```
cisco_wlc = {
    'device_type': 'cisco_ios',
    'ip': wlc_ip,
    'username': username,
    'password': password
}
net_connect = ConnectHandler(**cisco_wlc)
```

  - Create a while loop with a menu

```
# Text menu with 4 options that run each of the functions, or quit
while True:
    print("\nText Menu:")
    print("1. Show AP summary")
    print("2. Show client summary")
    print("3. Show CDP neighbors")
    print("4. Save run-config to file")
    print("5. Quit")

    choice = input("Enter choice: ")

    if choice == "1":
        show_ap_summary()
    elif choice == "2":
        show_client_summary()
    elif choice == "3":
        show_cdp_neighbors()
    elif choice == "4":
        save_config_to_file()
    elif choice == "5":
        net_connect.disconnect()
        break
    else:
        print("Invalid choice. Please choose again.")
```


Python - Using Netmiko - Hints: Task 3

- Example of functions

```
# Define functions
def show_ap_summary():
    print(net_connect.send_command("show ap summary"))

def show_client_summary():
    print(net_connect.send_command("show wireless client summary"))

def show_cdp_neighbors():
    print("WLC CDP neighbors:\n-------------------------------------")
    print(net_connect.send_command("show cdp neighbors"))
    print("AP CDP neighbors:\n-------------------------------------")
    print(net_connect.send_command("show ap cdp neighbors"))

def save_config_to_file():
    output=net_connect.send_command("show run")
    now = datetime.datetime.now()
    filename = f"{wlc_ip}-run-conf ({now.strftime('%Y-%m-%d %H:%M:%S')}).txt"
    with open(filename, 'w') as f:
        f.write(output)
    print(f"Config saved to {filename}")
```


Python - Using Netmiko - Example solution

- Here is an example solution
- Runtime output should be like this

```
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab40_Python_Netmiko$0_Python_Netmiko$ python netmiko-script.py
WLC IP: 192.168.10.9
Username: devnet-adm
Password:

Text Menu:
1. Show AP summary
2. Show client summary
3. Show CDP neighbors
4. Save run-config to file
5. Quit
Enter choice: 4
Config saved to 192.168.10.9-run-conf (2024-09-10 19:44:50).txt

Text Menu:
1. Show AP summary
2. Show client summary
3. Show CDP neighbors
4. Save run-config to file
5. Quit
Enter choice:
```

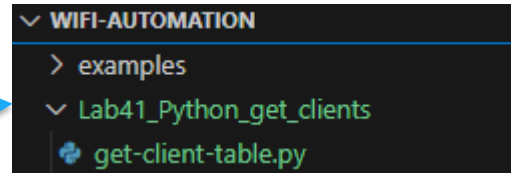
netmiko-script.py

```
import getpass
import datetime
from netmiko import ConnectHandler
wlc_ip = input('WLC IP: ')
username = input('Username: ')
password = getpass.getpass('Password: ')

# Use netmiko to connect to the WLC
cisco_wlc = {
    'device_type': 'cisco_ios',
    'ip': wlc_ip,
    'username': username,
    'password': password
}
net_connect = ConnectHandler(**cisco_wlc)

# Define functions
def show_ap_summary():
    print(net_connect.send_command("show ap summary"))

def show_client_summary():
    print(net_connect.send_command("show wireless client summary"))

def show_cdp_neighbors():
    print("WLC CDP neighbors:\n-----------------------------------")
    print(net_connect.send_command("show cdp neighbors"))
    print("AP CDP neighbors:\n-----------------------------------")
    print(net_connect.send_command("show ap cdp neighbors"))

def save_config_to_file():
    output=net_connect.send_command("show run")
    now = datetime.datetime.now()
    filename = f"{wlc_ip}-run-conf ({now.strftime('%Y-%m-%d
%H:%M:%S')}).txt"
    with open(filename, 'w') as f:
        f.write(output)
    print(f"Config saved to {filename}")


# Text menu with 4 options that run each of the functions, or quit
while True:
    print("\nText Menu:")
    print("1. Show AP summary")
    print("2. Show client summary")
    print("3. Show CDP neighbors")
    print("4. Save run-config to file")
    print("5. Quit")

    choice = input("Enter choice: ")

    if choice == "1":
        show_ap_summary()
    elif choice == "2":
        show_client_summary()
    elif choice == "3":
        show_cdp_neighbors()
    elif choice == "4":
        save_config_to_file()
    elif choice == "5":
        net_connect.disconnect()
        break
    else:
        print("Invalid choice. Please choose again.")
```


Get client table using RESTCONF - Intro

- Get client table from WLC using RESTCONF call
- Put client table in pandas dataframe
- Export to Excel
- Very similar to getting the AP table from Day 1
- You can try recreating this using the hints in this exercise
- Alternatively you can copy and modify the get-ap-table.py from day 1

Get client table using RESTCONF - Tasks



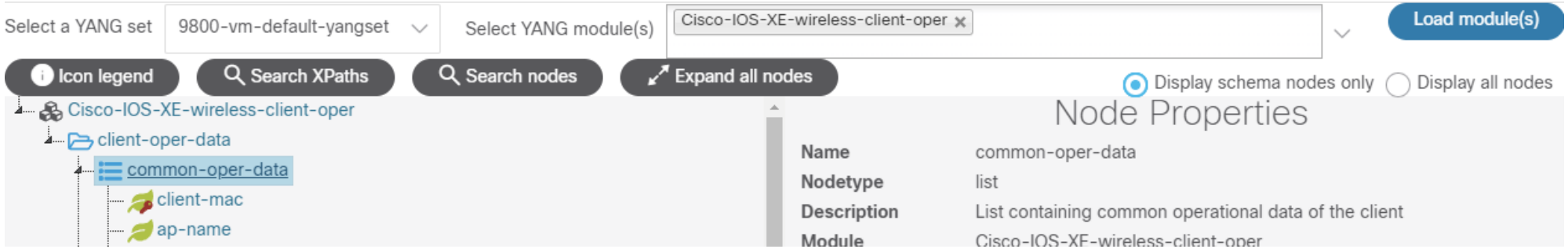
- Task 0: Set up the files and folders and VS Code
  - Create a working directory
  - Create a Python file called "get-client-table.py"
  - Activate the automation-venv 3.12.3 ('python-lab-venv': venv)

```
devnet-adm@ubuntu-devnet:~/wifi-automation/Lab40_Python_Netmiko$ source ~/automation-venv/bin/activate
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/ Lab40_Python_Netmiko$ python --version
```

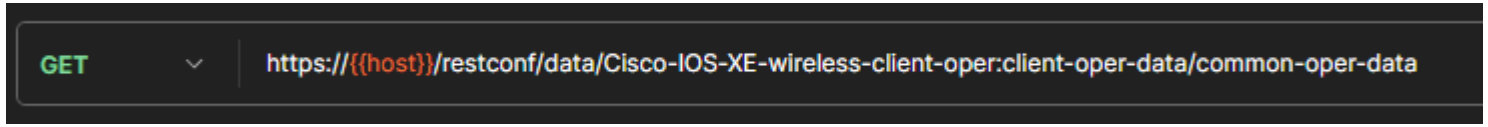
- Task 1: Use Postman to find and test an URL that gives you the connected clients
- Task 2: Write a Python script that
  - Ask for WLC IP, username and password
  - Connects to the WLC using RESTCONF with the URL created in Task 1
  - Print the results
- (optional) Task 3: Extend the script, so it will
  - Check if response status code is 200, if not, show the status code and message
  - Flatten the JSON of the response to a table, store in a pandas dataframe
  - Save the dataframe to Excel

Get client table using RESTCONF - Hints - Task 1

- Task 1: Use Postman to find and test an URL that gives you the connected clients
- Use YANG Suite, search for modules with the text "client"
- The module we will use in this exercise is "Cisco-IOS-XE-wireless-client-oper"



- Look at the request we created in the Day 1 exercise 9 "Exploring Postman"'. It contains the info that you need, and how to get there



Get client table using RESTCONF - Hints - Task 2

- Task 2: Write a Python script that
  - Ask for WLC IP, username and password
  - Connects to the WLC using RESTCONF with the URL created in Task 1
  - Print the results
- The input part of this script you can reuse from the netmiko script in Lab 40
- The RESTCONF connection will use the "requests" module. You should be able to get some hints from Postman, "output to code" and choose Python
- To create the URL including the WLC IP, you can use f-strings, like this:

```
url = f"https://{wlc}/restconf/data/Cisco-IOS-XE-wireless-client-oper:client-oper-data/common-oper-data"
```

- You can end/check this task by using a print statement after the requests call

```
response = requests.get(url, auth=(user, password), headers=headers, data=payload, verify=False)
print(response.json())
```

- Remember the imports (and maybe you need some pip installs along the way)

```
import requests
import getpass
```


Get client table using RESTCONF - Solution

- This example solution also includes the Task3 optional parts

get-client-table.py

```
import requests
import getpass
import pandas as pd

wlc = input("Enter WLC IP: ")
user = input("Enter user: ")
password = getpass.getpass("Enter password: ")
url = f"https://{wlc}/restconf/data/Cisco-IOS-XE-wireless-client-
oper:client-oper-data/common-oper-data"
payload = {}
headers = {
  'Accept': 'application/yang-data+json',
  'Content-Type': 'application/yang-data+json'
  }

response = requests.get(url, auth=(user, password), headers=headers,
data=payload, verify=False)

if (response.status_code==200):
    ap_table = pd.json_normalize(response.json()['Cisco-IOS-XE-wireless-
client-oper:client-oper-data'])
    print(ap_table)
    ap_table.to_excel('ap_table.xlsx')
else:
    print(f"Status code: {response.status_code}: {response.reason}")
```

- Create the API URL
- Do the actual RESTCONF call
- Check if response status code is 200, if not, show the status code and message
- Flatten the JSON of the response to a table, store in a pandas dataframe
- Save the dataframe to Excel

get-client-table.py U

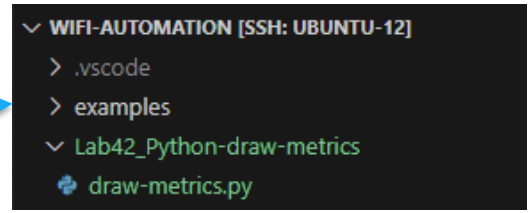
Lab41_Python_get_clients > get-client-table.py > ...

```
1   import requests
2   import getpass
3   import pandas as pd
4
5   wlc = input("Enter WLC IP: ")
6   user = input("Enter user: ")
7   password = getpass.getpass("Enter password: ")
8   url = f"https://{wlc}/restconf/data/Cisco-IOS-XE-wireless-client-oper:client-oper-data/common-oper-data"
9   payload = {}
10  headers = {
11    'Accept': 'application/yang-data+json',
12    'Content-Type': 'application/yang-data+json'
13    }
14
15  response = requests.get(url, auth=(user, password), headers=headers, data=payload, verify=False)
16
17  if (response.status_code==200):
18      ap_table = pd.json_normalize(response.json()['Cisco-IOS-XE-wireless-client-oper:client-oper-data'])
19      print(ap_table)
20      ap_table.to_excel('ap_table.xlsx')
21  else:
22      print(f"Status code: {response.status_code}: {response.reason}")
```


Get metrics and draw a graph - Intro

- Poll some metrics from WLC using RESTCONF
- Draw a graph of the metrics
- We will
  - Get data using RESTCONF
  - Put the data in an array
  - Use the library matplotlib for plotting the graph
- You will probably see that for pure monitoring, other tools (like Grafana) are way better ☺
- Notice: To get this plot to show on your screen, in Task 3 you will need to use a local installation of Python (on your laptop). This was not part of the preparations, so if you do not have it (or want to install now), just look at the provided screenshot

Get metrics and draw a graph - Tasks



- Task 0: Set up the files and folders and VS Code
  - Create a working directory "Lab42_Python-draw-metrics"
  - Create a Python file "draw-metrics.py"
  - Activate the automation-venv

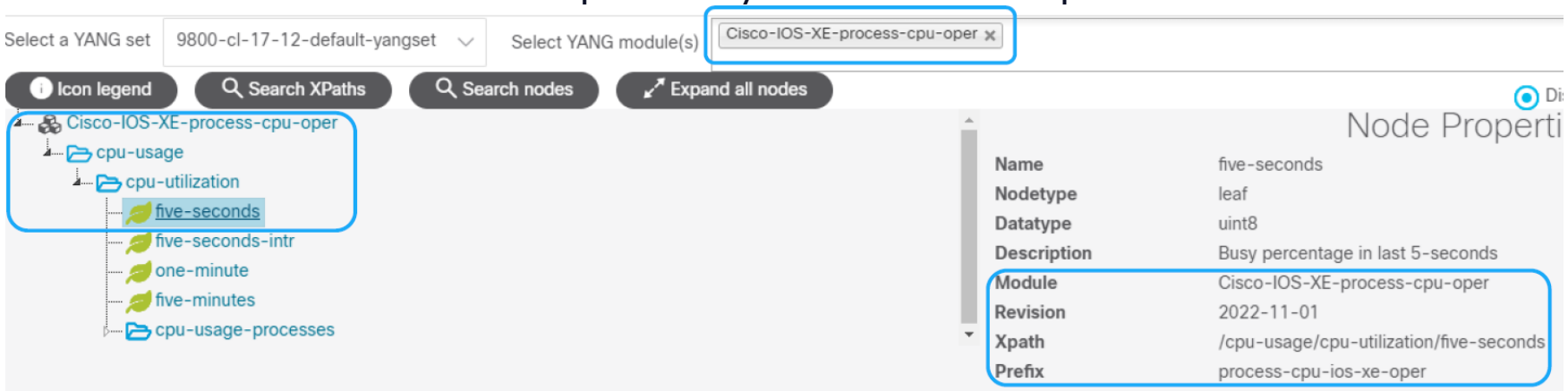
3.12.3 ('python-lab-venv': venv)

```
devnet-adm@ubuntu-devnet:~$ cd ~/wifi-automation/Lab42_Python-draw-metrics
devnet-adm@ubuntu-devnet:~/wifi-automation/Lab42_Python-draw-metrics$ source ~/automation-venv/bin/activate
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab42_Python-draw-metrics$ python --version
```

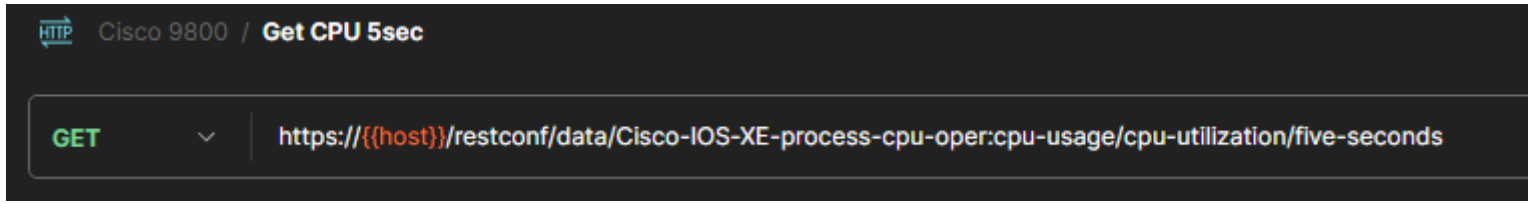
- Task 1: Use Postman to find and test an URL that gives you the 5-second CPU%
- Task 2: Write a Python script that
  - Ask for WLC IP, username and password
  - Connects to the WLC using RESTCONF with the URL created in Task 1
  - Loops and print the results every 2 seconds
- Task 3: Use matplotlib and get the script to plot the results
  - Store results in an array
  - Print the array using matplotlib
- (optional) Task 4: You can change the script to gather other data (#APs, #Clients, traffic)

Get metrics and draw a graph - Task 1 hints

- Task 1: Use Postman to find and test an URL that gives you the 5-second CPU%
- You can use YANG Suite to explore any module with "cpu"



- Then test in Postman



Get metrics and draw graph - Task 2 hints

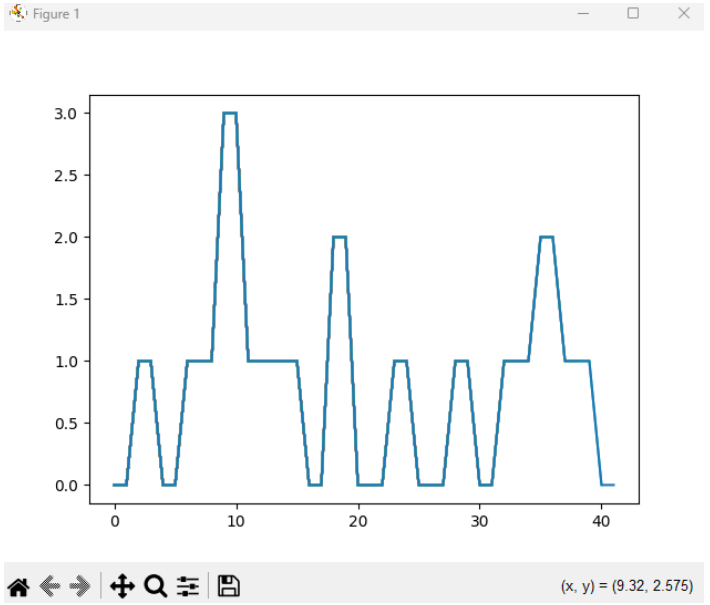
- Task 2: Write a Python script that
  - Ask for WLC IP, username and password
  - Connects to the WLC using RESTCONF with the URL created in Task 1
  - Loops and print the results every second
- You can use the script from Lab 41 as a base
  - Change the URL to the one you found in task 1
  - Remove the pandas dataframe and write to Excel parts
  - Start by printing response.json(), you can see how the JSON is built up
  - Then print response.json['Cisco-IOS-XE-process-cpu-oper:five-seconds']
  - Notice how you can reference keys inside a JSON using square brackets
  - Create a "while True:" loop of the last part, starting before the response = requests.... line
  - import time (at the top) and use time.sleep(1) inside the while-loop to pause for 1 second each iteration
  - By using a while loop with no exit condition, you exit the loop by pressing Ctrl+C
  - To suppress the InsecureRequestWarning, import urllib3 and use urllib3.disable_warnings()

draw-metrics.py (Task 2)

```
import time
import requests
import getpass
import urllib3

urllib3.disable_warnings()
wlc = input("Enter WLC IP: ")
user = input("Enter user: ")
password = getpass.getpass("Enter password: ")
url = f"https://{wlc}/restconf/data/Cisco-IOS-XE-process-cpu-oper:cpu-usage/cpu-utilization/five-seconds"
payload = {}
headers = {
  'Accept': 'application/yang-data+json',
  'Content-Type': 'application/yang-data+json'
  }

while True:
    response = requests.get(url, auth=(user, password), headers=headers, data=payload, verify=False)
    if (response.status_code==200):
        print(response.json()['Cisco-IOS-XE-process-cpu-oper:five-seconds'])
    else:
        print(f"Status code: {response.status_code}: {response.reason}")
    time.sleep(1)
```


Get metrics and draw a graph - Task 3 hints

- Task 3: Use matplotlib and get the script to plot the results
  - Store results in an array
  - Print the array using matplotlib
  - You will notice that "nothing happens". This is because matplotlib is running the plot on your Ubuntu server
  - To show this, you must store the script on your computer, and run using your local Python installation



Get metrics and draw a graph - Solution

Importing neccessary modules

Input section for IP, user, pass

Craft the RESTCONF URL and headers

Initialize a table with the first value

Initialize the plot

Endless loop

Request to WLC

Append the value to the table

Plot and draw the table

Alternative if status is not 200 "OK"

Pause before repeating the loop

draw-metrics.py

```
import time
import requests
import getpass
import urllib3
import matplotlib.pyplot as plt

urllib3.disable_warnings()
wlc = input("Enter WLC IP: ")
user = input("Enter user: ")
password = getpass.getpass("Enter password: ")
url = f"https://{wlc}/restconf/data/Cisco-IOS-XE-process-cpu-oper:cpu-usage/cpu-utilization/five-seconds"
payload = {}
headers = {
  'Accept': 'application/yang-data+json',
  'Content-Type': 'application/yang-data+json'
  }

cpu_values = [0]
plt.ion()
while True:
    response = requests.get(url, auth=(user, password), headers=headers, data=payload, verify=False)
    if (response.status_code==200):
        cpu_values.append(response.json()['Cisco-IOS-XE-process-cpu-oper:five-seconds'])
        plt.plot(cpu_values)
        plt.draw()
        plt.pause(1)
    else:
        print(f"Status code: {response.status_code}: {response.reason}")
    time.sleep(1)
```

lab42_Python-draw-metrics > draw-metrics.py > ...

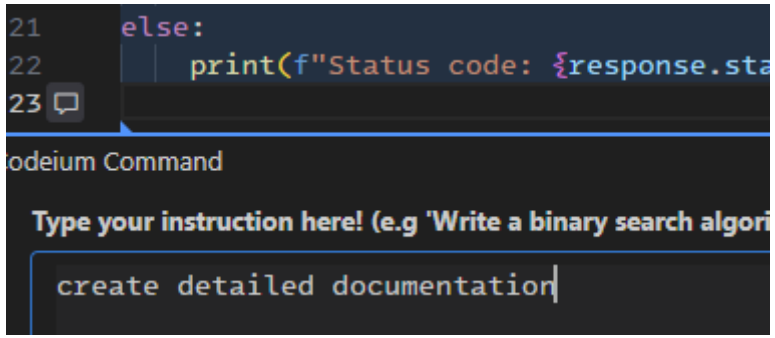
```python
import time
import requests
import getpass
import urllib3
import matplotlib.pyplot as plt

urllib3.disable_warnings()
wlc = input("Enter WLC IP: ")
user = input("Enter user: ")
password = getpass.getpass("Enter password: ")
url = f"https://{wlc}/restconf/data/Cisco-IOS-XE-process-cpu-oper:cpu-usage/cpu-utilization/five-seconds"
payload = {}
headers = {
  'Accept': 'application/yang-data+json',
  'Content-Type': 'application/yang-data+json'
  }

cpu_values = [0]
plt.ion()
while True:
    response = requests.get(url, auth=(user, password), headers=headers, data=payload, verify=False)
    if (response.status_code==200):
        cpu_values.append(response.json()['Cisco-IOS-XE-process-cpu-oper:five-seconds'])
        plt.plot(cpu_values)
        plt.draw()
        plt.pause(1)
    else:
        print(f"Status code: {response.status_code}: {response.reason}")
    time.sleep(1)
```

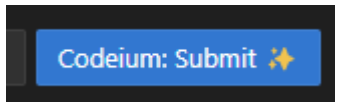

Get to know your AI companion - Intro and tasks

- Did you install Codeium? Or do you have GitHub Copilot? One very useful example of what they are really good at, is documenting your stuff!
- It can also help you get your ideas started. Beware that generative AI will not always make error-free code, or code that does what you intended. But it will often be a good start, and you can fix the rest yourself
- In this exercise, you will open some of the previous scripts, and check out different AI stuff, using Codeium
- Task 1: Document the code
- Task 2: Generate inline comments
- Task 3: See how generative AI would solve a previous task
- Task 4: Use AI to add or improve parts of the script

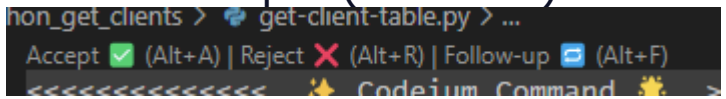
AI - Task 1 - Document the code

- Open one of the previous lab exercises. In this example we use the "get-client-table.py"
- Select all text in the .py file, click Ctrl+I, and write "create detailed documentation"

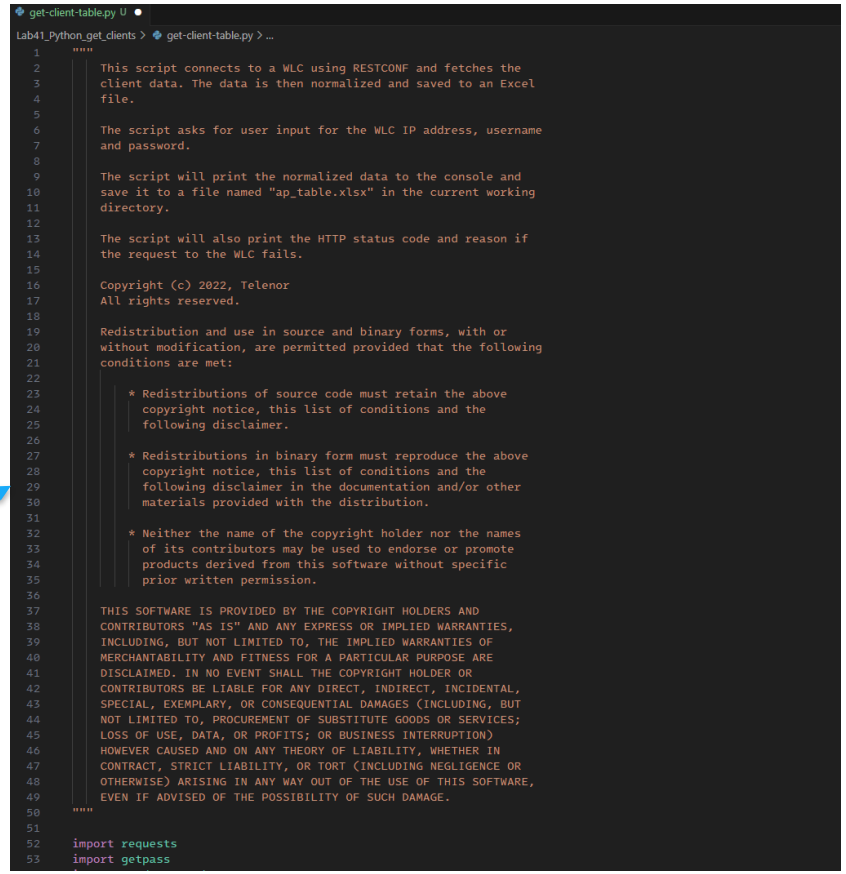




- Press "Accept" (or Alt+A) to accept

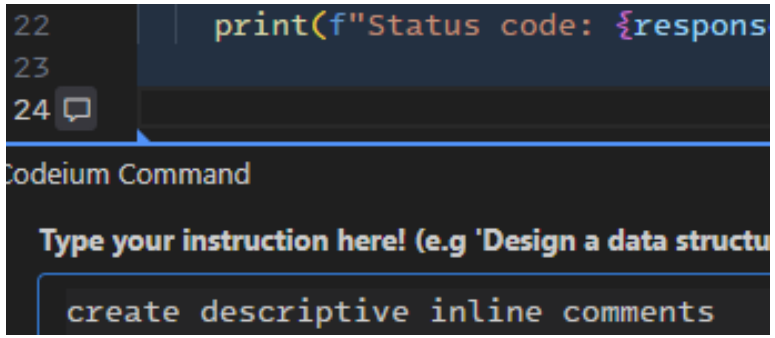


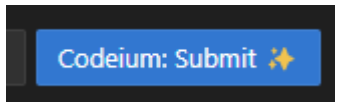
- The results will vary, when running this script now I got this
- The documentation is before the script



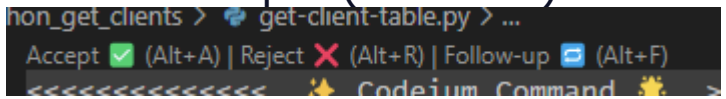
AI - Task 2 - Generate inline comments

- To improve readability, we can get AI to create inline comments
- Select all text in the .py file, click Ctrl+I, and write "create descriptive inline comments"





- Press "Accept" (or Alt+A) to accept

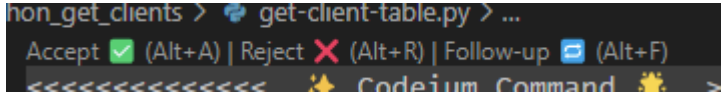


- The results will vary, when running this script now I got this
- The comments are inline

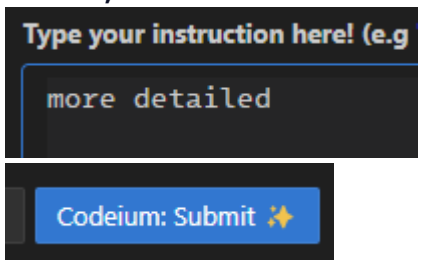
```python
import requests
import getpass
import pandas as pd

# Get input from user
wlc = input("Enter WLC IP: ")
user = input("Enter user: ")
password = getpass.getpass("Enter password: ")

# Construct URL for API query
url = f"https://{wlc}/restconf/data/Cisco-IOS-XE-wireless-client-oper:client-oper-data/common-oper-da

# Set headers for API query
payload = {}
headers = {
  'Accept': 'application/yang-data+json',  # Response format
  'Content-Type': 'application/yang-data+json'  # Request format
  }

# Make API query
response = requests.get(url, auth=(user, password), headers=headers, data=payload, verify=False)

# Check if query was successful
if (response.status_code==200):
    # Convert JSON response to DataFrame
    ap_table = pd.json_normalize(response.json()['Cisco-IOS-XE-wireless-client-oper:client-oper-data'
    # Print DataFrame
    print(ap_table)
    # Save DataFrame to Excel file
    ap_table.to_excel('ap_table.xlsx', index=False)
else:
    # Print error message
    print(f"Status code: {response.status_code}: {response.reason}")
```

AI - Task 2 - Follow-up

- You can also do follow-ups to alter or specify the results
- Press Ctrl+Z to undo, or do the last task again
- Instead of Accept, choose "Follow-up"



- Then, write "more detailed"

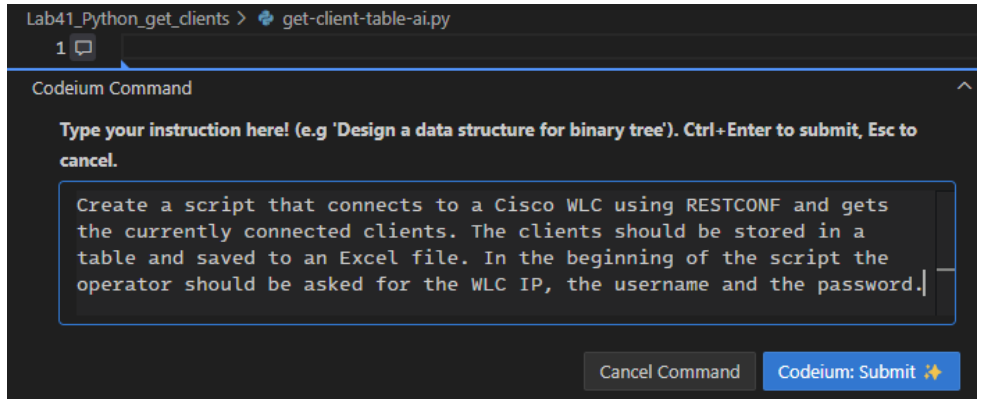


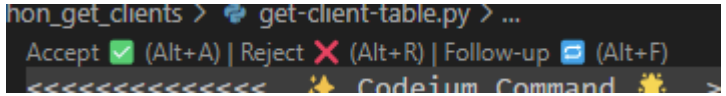
- Press "Accept" (or Alt+A) to accept
- The results will vary, when running this script now I got this
- The comments are a bit more detailed

```python
import requests
import getpass
import pandas as pd

# Ask user for WLC IP and credentials
wlc = input("Enter WLC IP: ")
user = input("Enter user: ")
password = getpass.getpass("Enter password: ")

# Construct API URL
url = f"https://{wlc}/restconf/data/Cisco-IOS-XE-wireless-client-oper:client-oper-data/common-oper-data"

# Set up API headers
payload = {}
headers = {
  # Specify JSON data format
  'Accept': 'application/yang-data+json',
  # Specify JSON data format for POST data
  'Content-Type': 'application/yang-data+json'
  }

# Send GET request to API
response = requests.get(url, auth=(user, password), headers=headers, data=payload, verify=False)

# Check if API request was successful
if (response.status_code==200):
    # Extract JSON data from API response
    ap_table = pd.json_normalize(response.json()['Cisco-IOS-XE-wireless-client-oper:client-oper-data'])
    # Print data to console
    print(ap_table)
    # Save data to Excel file
    ap_table.to_excel('ap_table.xlsx', index=False)
else:
    # Print error message if API request failed
    print(f"Status code: {response.status_code}: {response.reason}")
```


AI - Task 3 - Get AI to solve the task

- Check how AI would solve the task
- Create a new file "get-client-table-ai.py"
- In the empty file, Click Ctrl+I





- Press "Accept" (or Alt+A) to accept
- Notice it is pretty similar to your script? Probably because it use your files to train(?)

Lab41_Python_get_clients > get-client-table-ai.py > ...

```python
import requests
import getpass
import pandas as pd

# Ask user for WLC IP and credentials
wlc = input("Enter WLC IP: ")
user = input("Enter user: ")
password = getpass.getpass("Enter password: ")

# Construct API URL
url = f"https://{wlc}/restconf/data/Cisco-IOS-XE-wireless-client-oper:client-oper-data"

# Set up API headers
headers = {
  'Accept': 'application/yang-data+json',
  'Content-Type': 'application/yang-data+json'
  }

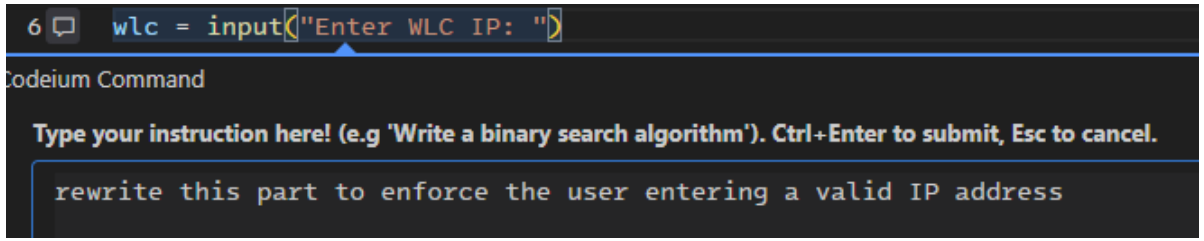
# Send GET request to API
response = requests.get(url, auth=(user, password), headers=headers, verify=False)

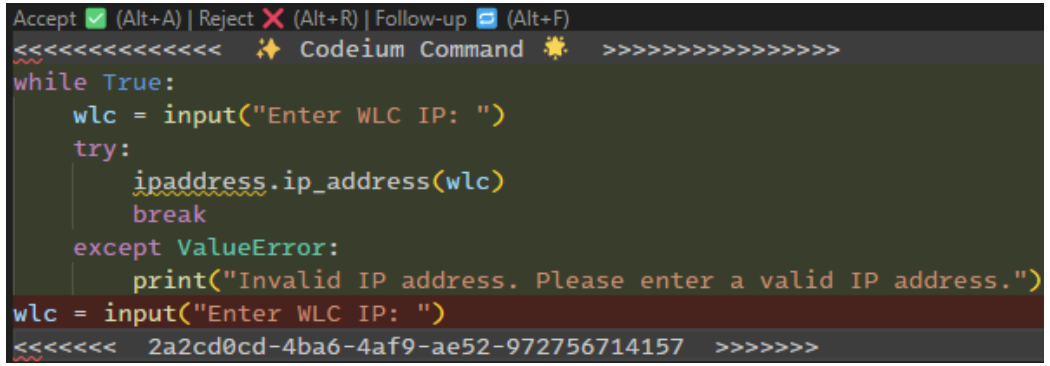
# Check if API request was successful
if (response.status_code==200):
    # Extract JSON data from API response
    clients = pd.json_normalize(response.json()['Cisco-IOS-XE-wireless-client-oper:client-oper-data'])
    # Save data to Excel file
    clients.to_excel('clients.xlsx', index=False)
else:
    # Print error message if API request failed
    print(f"Status code: {response.status_code}: {response.reason}")

```


AI - Task 4 - Use AI to add or improve parts

- Find some part to improve/change
- One example, enforce the user to input a valid IP address in the input part
- Mark the "Enter WLC IP" line, press Ctrl+I





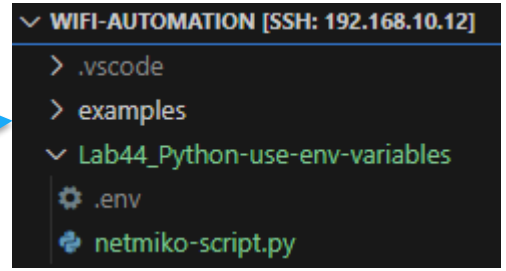
- Press "Accept" (or Alt+A) to accept
- Notice it adds "ipaddress" which is yellowed out by pylance. Try to import it.

```
import ipaddress
```


Use .env file for variables - Intro

- As an alternative to typing the WLC IP, username and password every time, they can be placed in a local file with the name ".env". By using this you can safely share the python script without including sensitive material in the script itself
- We will modify the "netmiko-script.py" from Lab 40, by adding support for reading the .env file. The contents of the script (except for those modifications), are already explained in Lab 40
- Remember that the .env file should be located in the same directory as you run the file from
- !!! Note !!! .env files are NOT synced by Git, they are included in a ".gitignore" file. This is by purpose, and is WHY we are using .env files instead of writing stuff in cleartext in the code. So if you look in the "solutions" folder, there will be no .env files in this or any other directory, you must create your own.

Use .env file for variables - Tasks



- Task 0: Set up the files and folders and VS Code
  - Create a working directory "Lab44_Python-use-env-variables"
  - Create a file ".env"
  - Copy the file "netmiko-script.py" from the previous exercise, or from the solutions folder
  - Activate the automation-venv 3.12.3 ('python-lab-venv': venv)

```
devnet-adm@ubuntu-devnet:~$ cd ~/wifi-automation/Lab44_Python-use-env-variables
devnet-adm@ubuntu-devnet:~/wifi-automation/Lab44_Python-use-env-variables$ source ~/automation-venv/bin/activate
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab44_Python-use-env-variables$ python --version
```

- Task 1: Take a look at https://pypi.org/project/python-dotenv/ and try implementing that in your script
- (optional) Task 2: Add a text showing which WLC you are connected to

.env

```
WLC_IP='192.168.10.{YOUR_WLC_IP}'
USERNAME='devnet-adm'
PASSWORD='ChangeMe2025!'
```

Use .env file for variables - Hints

Lab44_Python-use-env-variables > .env

```
1  WLC_IP='192.168.10.9'
2  USERNAME='devnet-adm'
3  PASSWORD='ChangeMe2024!'
```

- Enter the variables you need in the .env file
- Install dotenv for python, using PIP

```
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab44_Python-use-env-variables$ pip install python-dotenv
```

- !!! Notice the pip package name is not "dotenv" but "python-dotenv" !!!

- Import dotenv in the python file
- Load the env variables from file to a variable
- Modify the code to use the variables

```
4  from dotenv import dotenv_values
5
6  # Import variables from the .env file
7  env = dotenv_values('.env')
8
9  # Use netmiko to connect to the WLC
10 cisco_wlc = {
11     'device_type': 'cisco_ios',
12     'ip': env['WLC_IP'],
13     'username': env['USERNAME'],
14     'password': env['PASSWORD']
15 }
```

- There is a line a way down also, referencing the WLC IP. Change that as well to use the env

```
34     filename = f"{env['WLC_IP']}-run-conf ({now.strftime('%Y-%m-%d %H:%M:%S')}).txt"
```


Use .env file for variables - Example solution

- Here is an example solution
- Runtime output should be like this

```
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab44_Python-use-env-variables$ python netmiko-script.py

You are connected to WLC: 192.168.10.9
Menu:
1. Show AP summary
2. Show client summary
3. Show CDP neighbors
4. Save run-config to file
5. Quit
Enter choice: 4
Config saved to 192.168.10.9-run-conf (2024-09-10 19:44:50).txt

You are connected to WLC: 192.168.10.9
Menu:
1. Show AP summary
2. Show client summary
3. Show CDP neighbors
4. Save run-config to file
5. Quit
Enter choice:
```

- Notice how you do not have to type in IP, user and password every time
- The example output includes the optional task 2

netmiko-script.py

```
import getpass
import datetime
from netmiko import ConnectHandler
from dotenv import dotenv_values

# Import variables from the .env file
env = dotenv_values('.env')

# Use netmiko to connect to the WLC
cisco_wlc = {
    'device_type': 'cisco_ios',
    'ip': env['WLC_IP'],
    'username': env['USERNAME'],
    'password': env['PASSWORD']
}
net_connect = ConnectHandler(**cisco_wlc)

# Define functions
def show_ap_summary():
    print(net_connect.send_command("show ap summary"))

def show_client_summary():
    print(net_connect.send_command("show wireless client summary"))

def show_cdp_neighbors():
    print("WLC CDP neighbors:\n-------------------------------------")
    print(net_connect.send_command("show cdp neighbors"))
    print("AP CDP neighbors:\n-------------------------------------")
    print(net_connect.send_command("show ap cdp neighbors"))

def save_config_to_file():
    output=net_connect.send_command("show run")
    now = datetime.datetime.now()
    filename = f"{env['WLC_IP']}-run-conf ({now.strftime('%Y-%m-%d
%H:%M:%S')}).txt"
    with open(filename, 'w') as f:
        f.write(output)
    print(f"Config saved to {filename}")


# Text menu with 4 options that run each of the functions, or quit
while True:
    print('\nYou are connected to WLC: ' + env['WLC_IP'])
    print("Menu:")
    print("1. Show AP summary")
    print("2. Show client summary")
    print("3. Show CDP neighbors")
    print("4. Save run-config to file")
    print("5. Quit")

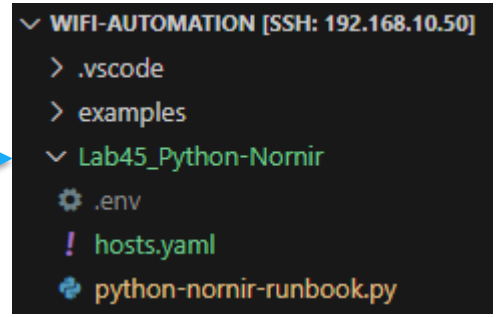
    choice = input("Enter choice: ")

    if choice == "1":
        show_ap_summary()
    elif choice == "2":
        show_client_summary()
    elif choice == "3":
        show_cdp_neighbors()
    elif choice == "4":
        save_config_to_file()
    elif choice == "5":
        net_connect.disconnect()
        break
    else:
        print("Invalid choice. Please choose again.")
```

Using Nornir - Intro

- Nornir is an automation framework written in Python
- You use Python code to use Nornir (compared to Ansible where you use YAML files)
- As with Ansible you will have
  - Inventory
  - Runbook (similar to Ansible "playbook")
  - Plugins
- There are pros and cons to every automation framework, test some variants and make up your own mind
- As a consultant, you might be bound to what your customer use
- Nornir vs Ansible pros: Speed, debugging, logging and "all-Python" approach
- Ansible vs Nornir pros: Idempotency, simplicity, YAML approach and community
- References:
  - https://nornir.readthedocs.io/en/latest/
  - https://developer.cisco.com/codeexchange/github/repo/neelimapp/nornir-sample-scripts/
  - https://theworldsgonemad.net/2021/nornir-tasks/

Using Nornir - Tasks



- Task 0: Set up the files and folders and VS Code
  - Create a working directory "Lab45_Python-Nornir"
  - Create a file ".env" (similar to Lab 44)
  - Create a file "hosts.yaml"
  - Create a file "python-nornir-runbook.py". You can reuse the dotenv parts from Lab 44
  - Activate the automation-venv

3.12.3 ('python-lab-venv': venv)

```
devnet-adm@ubuntu-devnet:~$ cd ~/wifi-automation/Lab45_Python-Nornir
devnet-adm@ubuntu-devnet:~/wifi-automation/Lab45_Python-Nornir$ source ~/automation-venv/bin/activate
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab45_Python-Nornir$ python --version
```

  - Install Nornir and some additional packages we will use, in your venv, using pip

```
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab45_Python-Nornir$ pip install nornir nornir-utils nornir-netmiko
```

- Task 1: Create inventory file
  - This exercise will use a very simple hosts.yaml file for inventory, with only the devices and the device type. Username and password will come from the .env file
- Task 2: Write a runbook that
  - Connects to all devices in the inventory
  - Do a "show ap summary"
  - Print the results on screen

Using Nornir - Hints

- In the .env file, we only put the username and password this time
- In the hosts.yml we put the WLC IP, you can easily scale this to multiple devices
- For the runbook "python_nornir_runbook.py"
  - Reuse the dotenv import and env = dotenv_values('.env') from Lab 44
  - Import some Nornir stuff that we need
  - Initialize Nornir (nr = InitNornir())
  - Set some Nornir defaults (nr.inventory.defaults.{blablabla})
    - username, get this from env['USERNAME']
    - password, get this from env['PASSWORD']
    - platform, just set this to the string "ios"
  - Run commands and save the output in "results"
    - ```
      nr.run(
          task = netmiko_send_command
          command_string = "show ap summary"
      )
      ```
  - Print "results" using Nornir: print_result(results)

.env

```
USERNAME='devnet-adm'
PASSWORD='ChangeMe2025!'
```

hosts.yaml

```
---
your_wlc:
   hostname: 192.168.10.{YOUR_WLC_IP}

shared_wlc:
   hostname: 192.168.10.9
```

```
from nornir import InitNornir  # Initialize Nornir
from nornir_utils.plugins.functions import print_result  # Print results
from nornir_netmiko.tasks import netmiko_send_command  # Send Netmiko command
```



```

Using Nornir - Example solution

- Here is an example solution
- Runtime output should be similar to this

```
(python-lab-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab45_Python-Nornir$ python python-nornir-runbook.py
netmiko_send_command*****************************************************************
* wlc ** changed : False ************************************************************
vvvv netmiko_send_command ** changed : False vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv INFO
Number of APs: 1

CC = Country Code
RD = Regulatory Domain

AP Name                          Slots AP Model             Ethernet MAC   Radio MAC      CC   RD   IP Address                                State        Location
---------------------------------------------------------------------------------------------------------------------------------------------------------------------
9120E-ekms                       2     C9120AXE-E           a4b4.392e.bfd0 4ca6.4d65.f940 NO   -E   192.168.10.182                            Registered   default location

^^^^ END netmiko_send_command ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

- As long as you use "hosts.yaml" you don't have to point to this anywhere, as it is the default inventory file that Nornir will look for, in the folder you run the script from
- You can use a dynamically created inventory, here are some examples
  - Service-Now can pass an inventory when you run stuff based on something you pass from there
  - You can use Netbox for inventory, and filter/pass/extract what you want from there. Like, if you get an action because an AP is down, run some checks on the switch and the WLC the AP was connected last. And some checks on all APs on the same location. Etc.

.env

```
USERNAME='devnet-adm'
PASSWORD='ChangeMe2025!'
```

hosts.yaml

```
---
your_wlc:
   hostname: 192.168.10.{YOUR_WLC_IP}

shared_wlc:
   hostname: 192.168.10.9
```

python-nornir-runbook.py

```
from dotenv import dotenv_values
from nornir import InitNornir  # Initialize Nornir
from nornir_utils.plugins.functions import print_result  # Print results
from nornir_netmiko.tasks import netmiko_send_command  # Send Netmiko command

# Import variables from the .env file
env = dotenv_values('.env')

# Initialize Nornir
nr = InitNornir()

# Set defaults for all devices
nr.inventory.defaults.username = env['USERNAME']  # Username from .env
nr.inventory.defaults.password = env['PASSWORD']  # Password from .env
nr.inventory.defaults.platform="ios"  # Platform set to Cisco IOS

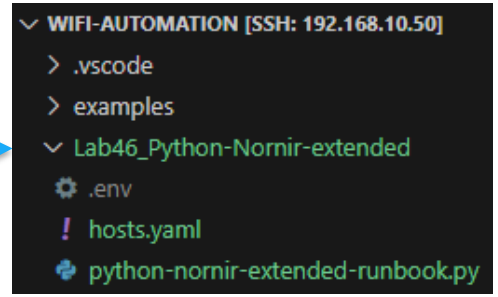
# Run command on all devices
results = nr.run(
    task=netmiko_send_command,  # Use the task "netmiko_send_command"
    command_string="show ap summary"  # Command to run
)

# Print results
print_result(results)
```

Nornir extended example - Intro

- In this task, we will combine our Nornir knowledge from the previous lab, with the script we created in the first Python exercise using Netmiko, as well as the Python exercise using .env file
- In short, the single-WLC script using Netmiko will be changed so it runs on all WLCs in a Nornir inventory
- You can easily expand the inventory to
  - Your own WLC
  - The shared WLC
  - Other students' WLCs that you have access to

Nornir extended example - Tasks



- Task 0: Set up the files and folders and VS Code
  - Create a working directory "Lab46_Python-Nornir-extended"
  - Create a file ".env" (copy from Lab 45)
  - Create a file "hosts.yaml" (copy from Lab 45)
  - Create a file "python-nornir-extended-runbook.py". Start by copy-pasting the "netmiko-script.py" from Lab 44
  - Activate the automation-venv

3.12.3 ('python-lab-venv': venv)

```
devnet-adm@ubuntu-devnet:~$ cd ~/wifi-automation/Lab46_Python-Nornir-extended
devnet-adm@ubuntu-devnet:~/wifi-automation/Lab46_Python-Nornir-extended$ source ~/automation-venv/bin/activate
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab46_Python-Nornir-extended$ python --version
```

  - Nornir and required packages should already be installed from the previous lab

- Task 1: Merge the necessary Nornir components from Lab 45 into the new runbook
  - Add Nornir imports, set Nornir defaults, initialize Nornir
  - Remove the Netmiko imports, definition of cisco_wlc object and initializing net_connect = ConnectHandler()
- Task 2: Rewrite the functions that are called from the menu, to be run using nornir_netmiko and use the Nornir inventory
  - All lines referring to "net_connect" will need to be changed
- Task 3: Rewrite the menu with "You are connected to", to show all devices from the inventory that you will issue the commands to

Nornir extended example - Task 1 hints

- In the .env file, we only put the username and password
- In the hosts.yml we put the WLC IP, you can easily scale this to multiple devices
- Add the Nornir imports, remove the Netmiko imports

```python
import datetime
from dotenv import dotenv_values
from nornir import InitNornir  # Initialize Nornir
from nornir_utils.plugins.functions import print_result  # Print results
from nornir_netmiko.tasks import netmiko_send_command  # Send Netmiko command
```

- Initialize Nornir and set the defaults.

```python
# Initialize Nornir
nr = InitNornir()

# Set defaults for all devices
nr.inventory.defaults.username = env['USERNAME']  # Username from .env
nr.inventory.defaults.password = env['PASSWORD']  # Password from .env
nr.inventory.defaults.platform="ios"  # Platform set to Cisco IOS

# Define functions
```

Remove the Netmiko initialization

```python
# Use netmiko to connect to the WLC
cisco_wlc = {
    'device_type': 'cisco_ios',
    'ip': env['WLC_IP'],
    'username': env['USERNAME'],
    'password': env['PASSWORD']
}
net_connect = ConnectHandler(**cisco_wlc)

# Define functions
```

.env

```
USERNAME='devnet-adm'
PASSWORD='ChangeMe2025!'
```

hosts.yaml

```
---
your_wlc:
   hostname: 192.168.10.{YOUR_WLC_IP}

shared_wlc:
   hostname: 192.168.10.9
```


Nornir extended example - Task 2 hints

- Change all lines referring to "net_connect". That will be in each of the 4 functions
  - show_ap_summary()

    Change:
    ```
    print(net_connect.send_command("show ap summary"))
    ```

    To:
    ```
    results = nr.run(
        task=netmiko_send_command,
        command_string="show ap summary"
    )
    print_result(results)
    ```
  - show_client_summary()

    Do the same changes as in previous function
  - show_cdp_neighbors()

    Do the same changes as in previous functions
  - save_config_to_file()

    Firstly, do the same changes as in previous functions:
    - In the existing script, we saved the output from netmiko to "output"
      ```
      output=net_connect.send_command("show run")
      ```
    - Now, change this to the same format as the previous functions, saving the results in "results". Exactly the same as the previous functions. The rest of this function (saving of the file) will follow on next slide

Nornir extended example - Task 2 hints continued

- Continuing on the save_config_to_file() function, we are now ready to save the results to a file
  - The results object is actually an object of type "AggregatedResult", containing results for each object that tasks has been run on, as a "MultiResult" object. Each MultiResult object is accessed by the name of the host. So for our example host file you would have two MultiResult objects:

```
results['your_wlc']       # Object of type MultiResult
results['shared_wlc']     # Object of type MultiResult
```

  - Each of these contains a list with the results of each task run on that object, starting from 0. So with one task run for the "your_wlc" host, the output (e.g. the "show run") will be stored in:

```
results['your_wlc'][0]              # Output of the first task. Use str() to change this to a text object.
```

  - This object will also have a status (boolean value) telling if the task failed. We will use this to only save the run-config if the command has NOT failed. Failed tasks can be if login failed, host unreachable, etc.

```
results['your_wlc'][0].failed             # Boolean value, True or False
```

  - So, the rest of the function can be something like this:

```
    for host in nr.inventory.hosts:                # Loop over all hosts in the Nornir inventory
        if results[host][0].failed:                # Check if the first task have status failed, if failed print an error message
            print(f'Save run-config for {host} ({nr.inventory.hosts[host].hostname}) failed. See nornir.log for details.')
        else:                                      # Else (if the task is NOT failed), run the file writing part
            now = datetime.datetime.now()          # Get the current timestamp
            filename = f"{nr.inventory.hosts[host].hostname}-run-conf ({now.strftime('%Y-%m-%d %H:%M:%S')}).txt" # Make a filename
            with open(filename, 'w') as f:         # Write the actual file
                f.write(str(results[host][0]))
            print(f"Config saved to {filename}")
```

- Reference and more info: https://nornir.readthedocs.io/en/v2.5.0/tutorials/intro/task_results.html

Nornir extended example - Task 3 hints

- Task 3 was to rewrite the menu elements. The old menu started by showing the (one) WLC you were connected to. For the Nornir variant we can be connected to lots of WLCs, so we have to adjust the menu accordingly

Old WLC connection info:

```
    print('\nYou are connected to WLC: ' + env['WLC_IP'])
```

New WLC connection info:

```
    print('\nYou are connected to these devices: ')
    for host in nr.inventory.hosts:
        print(f'-> {host}: {nr.inventory.hosts[host].hostname}')
```

In this example, you loop over the hosts found in the Nornir inventory (nr.inventory.hosts), and call the element in each of your loop interations "host"

In the next line you use an f-string, and print the text of the host, along which the hostname found in the Nornir inventory, in our case the "hostname" will be the IP address of the device

This info corresponds to what you have written in the hosts.yaml file

```
---
your_wlc:
   hostname: 192.168.10.10

shared_wlc:
   hostname: 192.168.10.9
```

Output in my example

```
You are connected to these devices:
-> your_wlc: 192.168.10.10
-> shared_wlc: 192.168.10.9
Menu:
```


Nornir extended example - Example solution

- Here is an example solution
- Runtime output should be similar to this

```
(python-lab-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab46_Python-Nornir-extended$ python python-nornir-extended-runbook.py

You are connected to these devices:
-> your_wlc: 192.168.10.10
-> shared_wlc: 192.168.10.9
Menu:
1. Show AP summary
2. Show client summary
3. Show CDP neighbors
4. Save run-config to file
5. Quit
Enter choice: 4
Saving config to file...
Config saved to 192.168.10.10-run-conf (2024-10-05 22:15:42).txt
Save run-config for shared_wlc (192.168.10.9) failed. See nornir.log for details.

You are connected to these devices:
-> your_wlc: 192.168.10.10
-> shared_wlc: 192.168.10.9
Menu:
1. Show AP summary
2. Show client summary
3. Show CDP neighbors
4. Save run-config to file
5. Quit
Enter choice:
```

.env

```
USERNAME='devnet-adm'
PASSWORD='ChangeMe2025!'
```

hosts.yaml

```
---
your_wlc:
   hostname: 192.168.10.{YOUR_WLC_IP}

shared_wlc:
   hostname: 192.168.10.9
```

python-nornir-extended-runbook.py

```
import datetime
from dotenv import dotenv_values
from nornir import InitNornir  # Initialize Nornir
from nornir_utils.plugins.functions import print_result  # Print results
from nornir_netmiko.tasks import netmiko_send_command  # Send Netmiko command

# Import variables from the .env file
env = dotenv_values('.env')

# Initialize Nornir
nr = InitNornir()

# Set defaults for all devices
nr.inventory.defaults.username = env['USERNAME']  # Username from .env
nr.inventory.defaults.password = env['PASSWORD']  # Password from .env
nr.inventory.defaults.platform="ios"  # Platform set to Cisco IOS

# Define functions
def show_ap_summary():
    results = nr.run(
        task=netmiko_send_command,
        command_string="show ap summary"
    )
    print_result(results)

def show_client_summary():
    results = nr.run(
        task=netmiko_send_command,
        command_string="show wireless client summary"
    )
    print_result(results)

def show_cdp_neighbors():
    print("WLC CDP neighbors:\n----------------------------------")
    results = nr.run(
        task=netmiko_send_command,
        command_string="show cdp neighbors"
    )
    print_result(results)
    print("AP CDP neighbors:\n----------------------------------")
    results = nr.run(
        task=netmiko_send_command,
        command_string="show ap cdp neighbors"
    )
    print_result(results)

def save_config_to_file():
    print("Saving config to file...")
    results = nr.run(
        task=netmiko_send_command,
        command_string="show run"
    )
    for host in nr.inventory.hosts:
        if results[host][0].failed:
            print(f"Save run-config for {host} ({nr.inventory.hosts[host].hostname}) failed. See nornir.log for details.")
        else:
            now = datetime.datetime.now()
            filename = f"{nr.inventory.hosts[host].hostname}-run-conf ({now.strftime('%Y-%m-%d %H:%M:%S')}).txt"
            with open(filename, 'w') as f:
                f.write(str(results[host][0]))
            print(f"Config saved to {filename}")


# Text menu with 4 options that run each of the functions, or quit
while True:
    print('\nYou are connected to these devices: ')
    for host in nr.inventory.hosts:
        print(f'-> {host}: {nr.inventory.hosts[host].hostname}')
    print("Menu:")
    print("1. Show AP summary")
    print("2. Show client summary")
    print("3. Show CDP neighbors")
    print("4. Save run-config to file")
    print("5. Quit")

    choice = input("Enter choice: ")

    if choice == "1":
        show_ap_summary()
    elif choice == "2":
        show_client_summary()
    elif choice == "3":
        show_cdp_neighbors()
    elif choice == "4":
        save_config_to_file()
    elif choice == "5":
        break
    else:
        print("Invalid choice. Please choose again.")
```

In-depth: Grafana / TIG-stack

- This part of the Day 2 labs will focus on using Grafana, Telegraf and InfluxDB
- The following lab exercises are included in this part
  - Lab Exercise #30: Grafana – Syslog from WLC and APs

Lab Exercise #30: Grafana - Syslog

- We will use rsyslog that is usually enabled by default on our Ubuntu VM
  - Check if rsyslog is installed and enabled

```
devnet-adm@ubuntu-devnet:~$ sudo systemctl status rsyslog
● rsyslog.service - System Logging Service
     Loaded: loaded (/usr/lib/systemd/system/rsyslog.service; enabled; preset: enabled)
     Active: active (running) since Fri 2024-09-06 13:43:01 UTC; 3 days ago
TriggeredBy: ● syslog.socket
       Docs: man:rsyslogd(8)
(...cut for brevity...)
```

  - (Conditional) If rsyslog is not installed, run these commands

```
devnet-adm@ubuntu-devnet:~$ sudo apt update && sudo apt install rsyslog
devnet-adm@ubuntu-devnet:~$ sudo systemctl start rsyslog
```

- Create a new config file for rsyslog

```
devnet-adm@ubuntu-devnet:~$ sudo mv /etc/rsyslog.conf /etc/rsyslog.conf.bak
devnet-adm@ubuntu-devnet:~$ sudo nano /etc/rsyslog.conf
# Paste the following lines to the file
```

- Then restart rsyslog with the new config

```
devnet-adm@ubuntu-devnet:~$ sudo systemctl restart rsyslog
```

cli-playbook.yaml

```
# /etc/rsyslog.conf configuration file for rsyslog

#################
#### MODULES ####
#################

module(load="imuxsock") # provides support for local system logging
module(load="imudp")    # provides UDP syslog reception
module(load="imtcp")    # provides TCP syslog reception
module(load="imklog")   # provides kernel logging support

# Provides UDP syslog reception
input(type="imudp" port="514")

# Provides TCP syslog reception
input(type="imtcp" port="514")

###########################
#### GLOBAL DIRECTIVES ####
###########################

$ActionFileDefaultTemplate RSYSLOG_TraditionalFileFormat
$RepeatedMsgReduction on
$FileOwner syslog
$FileGroup adm
$FileCreateMode 0640
$DirCreateMode 0755
$Umask 0022
$PrivDropToUser syslog
$PrivDropToGroup syslog
$WorkDirectory /var/spool/rsyslog
$ActionQueueType LinkedList
$ActionQueueFileName srvrfwd
$ActionResumeRetryCount -1
$ActionQueueSaveOnShutdown on

# Filter messages based on source IP
if $fromhost-ip startswith '192.168.10.' then {
    *.* @@(o)127.0.0.1:6514;RSYSLOG_SyslogProtocol23Format
}
else {
    stop
}

# Include all config files in /etc/rsyslog.d/
$IncludeConfig /etc/rsyslog.d/*.conf
```

Lab Exercise #30: Grafana - Syslog

- We need to make sure to change the docker-compose file.
  - Telegraf needs to listen to the correct ports
    - Add "6514:6514/tcp"

docker-compose.yml

```
telegraf:
   container_name: telegraf
   image: telegraf:1.32.0
   restart: always
   ports:
     - "6514:6514/tcp"
     - 57000:57000
```

```
devnet-adm@ubuntu-devnet:~$ cd ~/tig-stack/
devnet-adm@ubuntu-devnet:~/tig-stack$ nano docker-compose.yml
```

- Bring the containers down, then up, to read the new docker-compose.yml

```
devnet-adm@ubuntu-devnet:~/tig-stack$ docker compose down
[+] Running 4/4
 ✓ Container grafana          Removed
 ✓ Container telegraf         Removed
 ✓ Container influxdb         Removed
 ✓ Network tig-stack_default  Removed
devnet-adm@ubuntu-devnet:~/tig-stack$ docker compose up -d
[+] Running 4/4
 ✓ Container grafana          Created
 ✓ Container telegraf         Started
 ✓ Container influxdb         Started
 ✓ Network tig-stack_default  Started
devnet-adm@ubuntu-devnet:~/tig-stack$ docker ps | grep "telegraf"
73aafadd0c91   telegraf:1.32.0          "/entrypoint.sh tele…"   7 minutes ago   Up 6 minutes   8092/udp, 0.0.0.0:6514->6514/tcp,
:::6514->6514/tcp, 0.0.0.0:57000->57000/tcp, :::57000->57000/tcp, 8125/udp, 8094/tcp
```

Should show the new TCP port we are listening to

Lab Exercise #30: Grafana - Syslog

- We also need to edit the telegraf.conf file from day 1.
  - We will update with a new input-plugin, syslog and a new output plugin so the syslog ends up in the same databases
- We will use tags in the telegraf file to make sure the data ends up in the correct docker.

```
devnet-adm@ubuntu-devnet:~$ cd ~/tig-stack/
devnet-adm@ubuntu-devnet:~/tig-stack$ nano telegraf/conf/telegraf.conf
```

- Paste in the follwing under output plugins and input plugins
- Restart the telegraf docker

```
devnet-adm@ubuntu-devnet:~$ docker restart telegraf
```

- **(Optional)** make it more secure, by making a different token by Telegraf for each output
  - We will use the standard admin token token for all RW in this deepdive.☺

telegraf.conf

```
###############################################################################
#                            OUTPUT PLUGINS                                   #
###############################################################################


# Output for Syslog data
[[outputs.influxdb_v2]]
urls = [ "http://influxdb:8086" ]
  token = "${INFLUXDB_ADMIN_TOKEN}"
  organization = "${INFLUXDB_ORG}"
  bucket = "${INFLUXDB_BUCKET}"
  bucket = "syslog-bucket"
  tagexclude = ["c9800-bucket"]
  [outputs.influxdb_v2.tagpass]
    influxdb_database = ["syslog-bucket"]
  [outputs.influxdb_v2.namedrop]
    namedrop = ["Cisco-IOS-XE-wireless*"]



###############################################################################
#                            INPUT PLUGINS                                    #
###############################################################################


# Syslog input configuration
[[inputs.syslog]]
  ## Specify the address to listen on
  server = "tcp://0.0.0.0:6514"  # Bind to all interfaces on port 6514 for TCP
  ## Alternatively, you can listen on UDP
  # server = "udp://0.0.0.0:6514"

  ## Parse data from RFC5424 or RFC3164 format
  best_effort = true
  ## Data format to extract metrics from syslog messages
  data_format = "syslog"
 [inputs.syslog.tags]
  influxdb_database = "syslog-bucket"
```

Lab Exercise #30: Grafana - Syslog

- Log into your lab_WLC and add the following config
  - Change IP to your pod-WLC-IP

```
9800CL-Proxmox(config)# conf t
!
9800CL-Proxmox(config)# logging host <your_Ubuntu_IP>
!
9800CL-Proxmox(config)# wireless client syslog-detailed
!
9800CL-Proxmox(config)# logging syslog-events informational
!
9800CL-Proxmox(config)# ap profile default-ap-profile
9800CL-Proxmox(config-ap-profile)#syslog host <your_Ubuntu_IP>
9800CL-Proxmox(config-ap-profile)#syslog level informational
9800CL-Proxmox(config-ap-profile)#exit
9800CL-Proxmox(config)#
```

The following are the Syslog server logging levels:

- **emergencies** –Signifies severity 0. Implies that the system is not usable.
- **alerts** –Signifies severity 1. Implies that an immediate action is required.
- **critical** –Signifies severity 2. Implies critical conditions.
- **errors** –Signifies severity 3. Implies error conditions.
- **warnings** –Signifies severity 4. Implies warning conditions.
- **notifications** –Signifies severity 5. Implies normal but significant conditions.
- **informational** –Signifies severity 6. Implies informational messages.
- **debugging** –Signifies severity 7. Implies debugging messages.

Note To know the number of Syslog levels supported, you need to select a Syslog level. Once a Syslog level is selected, all the levels below it are also enabled.

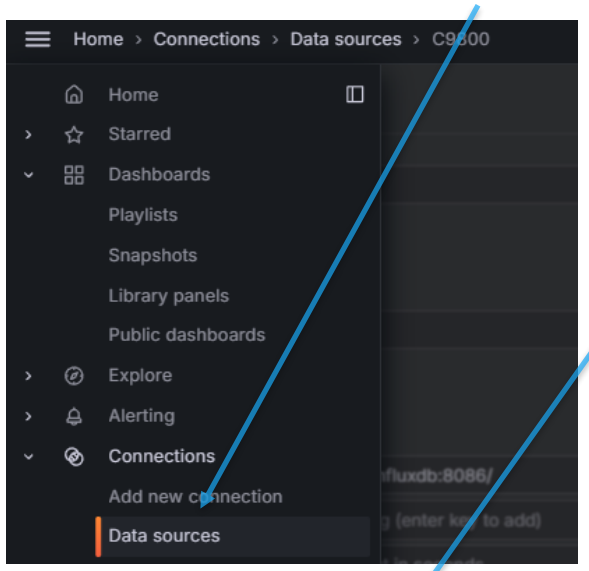
If you enable *critical* Syslog level then all levels below it are also enabled. So, all three of them, namely, *critical*, *alerts*, and *emergencies* are enabled.

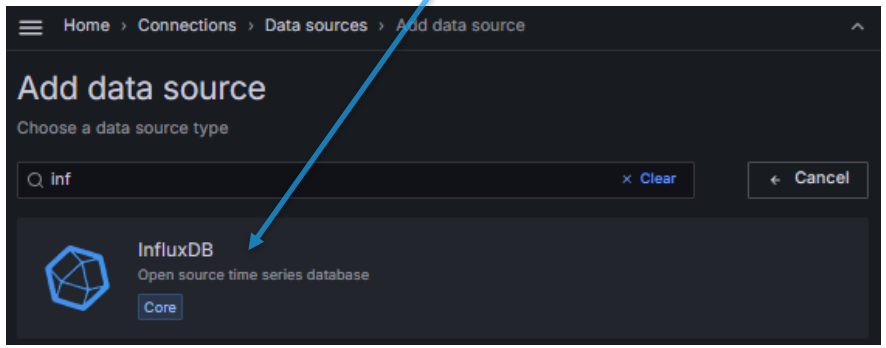
- Create syslog of all configuration commands

```
9800CL-Proxmox(config)# archive
9800CL-Proxmox(config-archive)# log config
9800CL-Proxmox(config-archive-log-cfg)# logging enable
9800CL-Proxmox(config-archive-log-cfg)# logging size 200
9800CL-Proxmox(config-archive-log-cfg)# notify syslog contenttype plaintext
9800CL-Proxmox(config-archive-log-cfg)# end
9800CL-Proxmox# wr
```

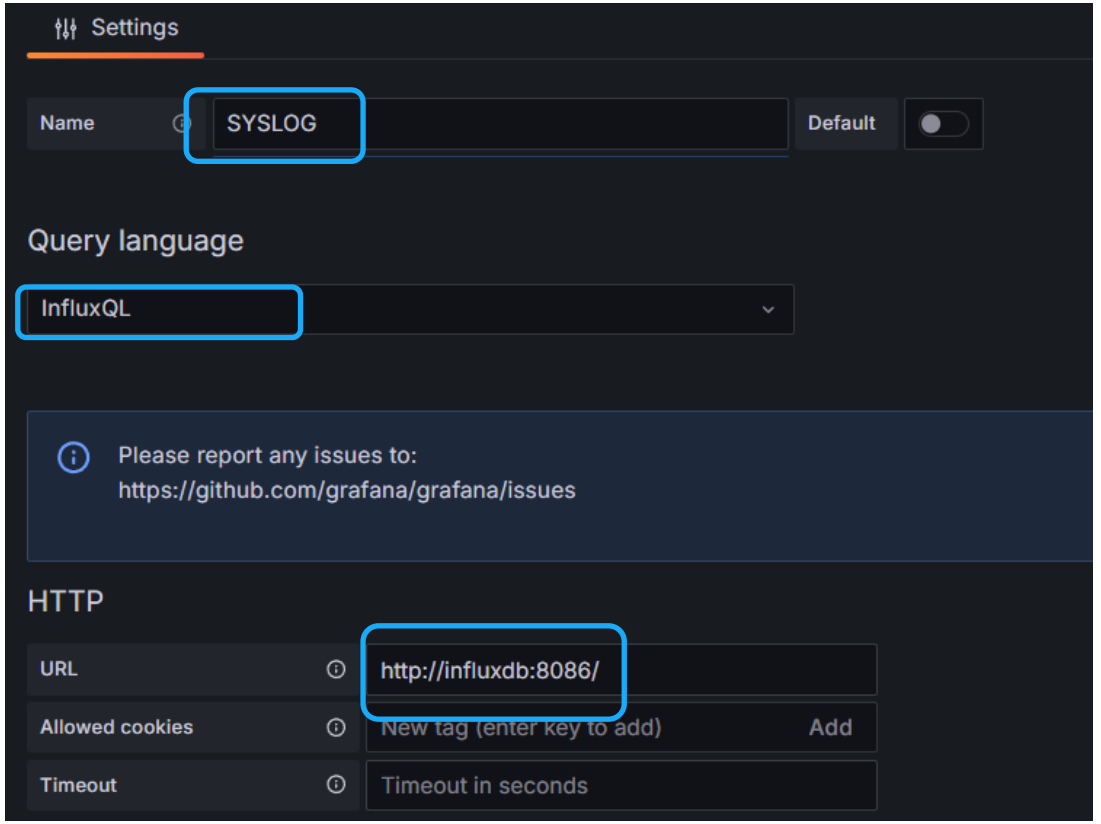
Lab Exercise #30: Grafana - Syslog

- Add a new data source -> InfluxDB



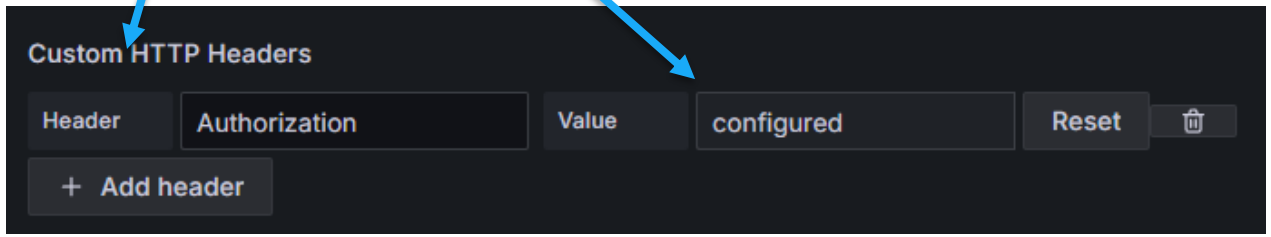


- Set a name for your database, f.ex syslog
- Set Query language to InfluxQL
- URL: http://influxdb:8086

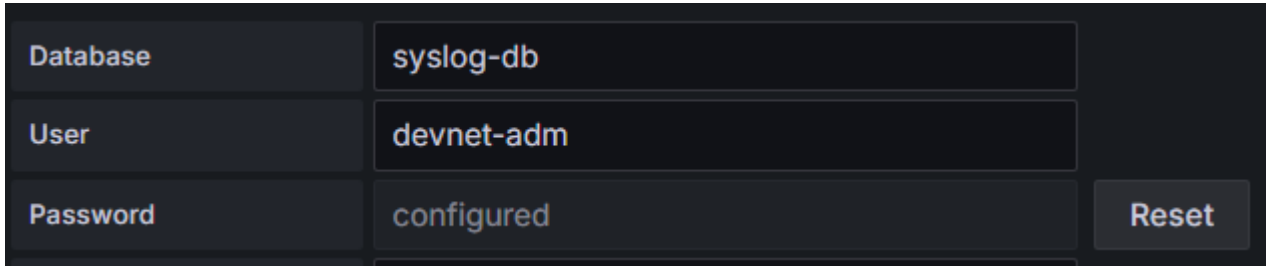


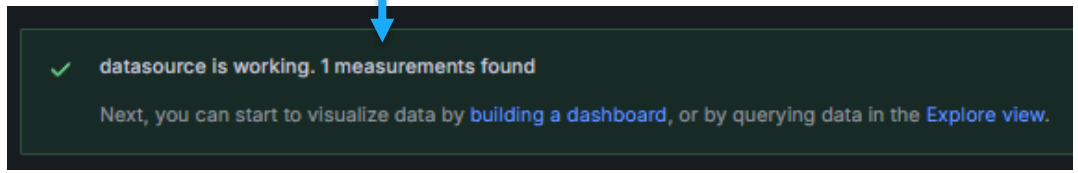
Lab Exercise #30: Grafana - Syslog

- Use the token from your lab_10 notes (grafana RO token)
  - Add a the custom HTTP Headers
    - Header: Authorization
    - Value: **Token <API-TOKEN>**
    - **There is a space between Token and <token>**



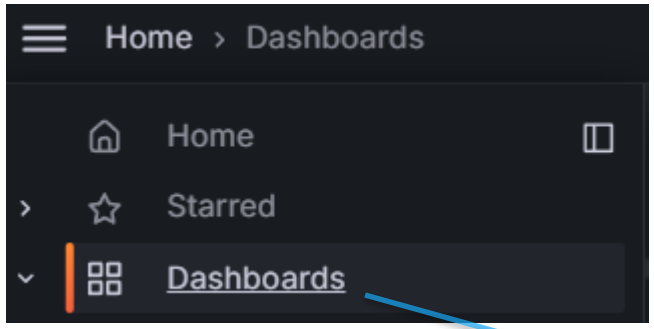
  - Database: syslog-db
    - User/Pass: devnet-adm / ChangeMe2025!
    - Value: **Token <API-TOKEN>**

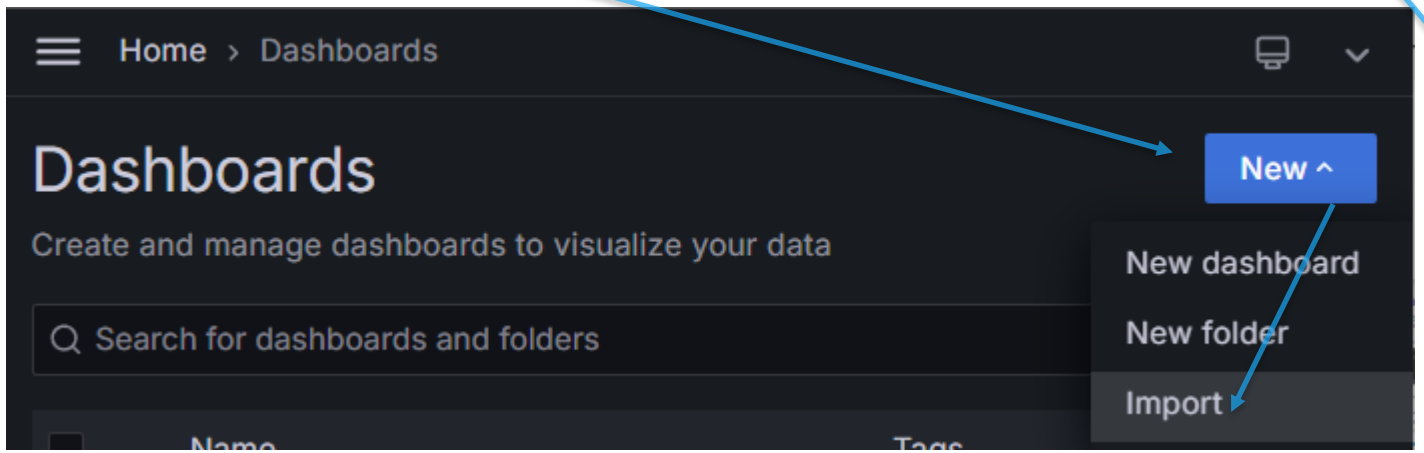




Lab Exercise #30: Grafana - Syslog

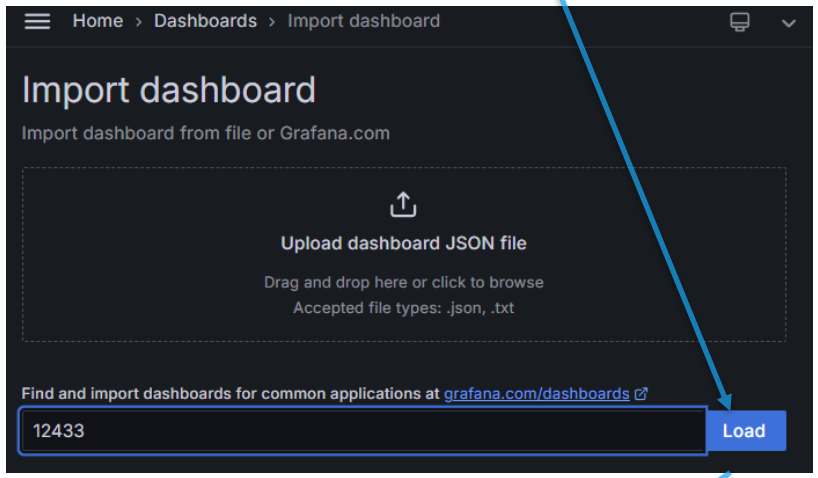
- Import dashboard



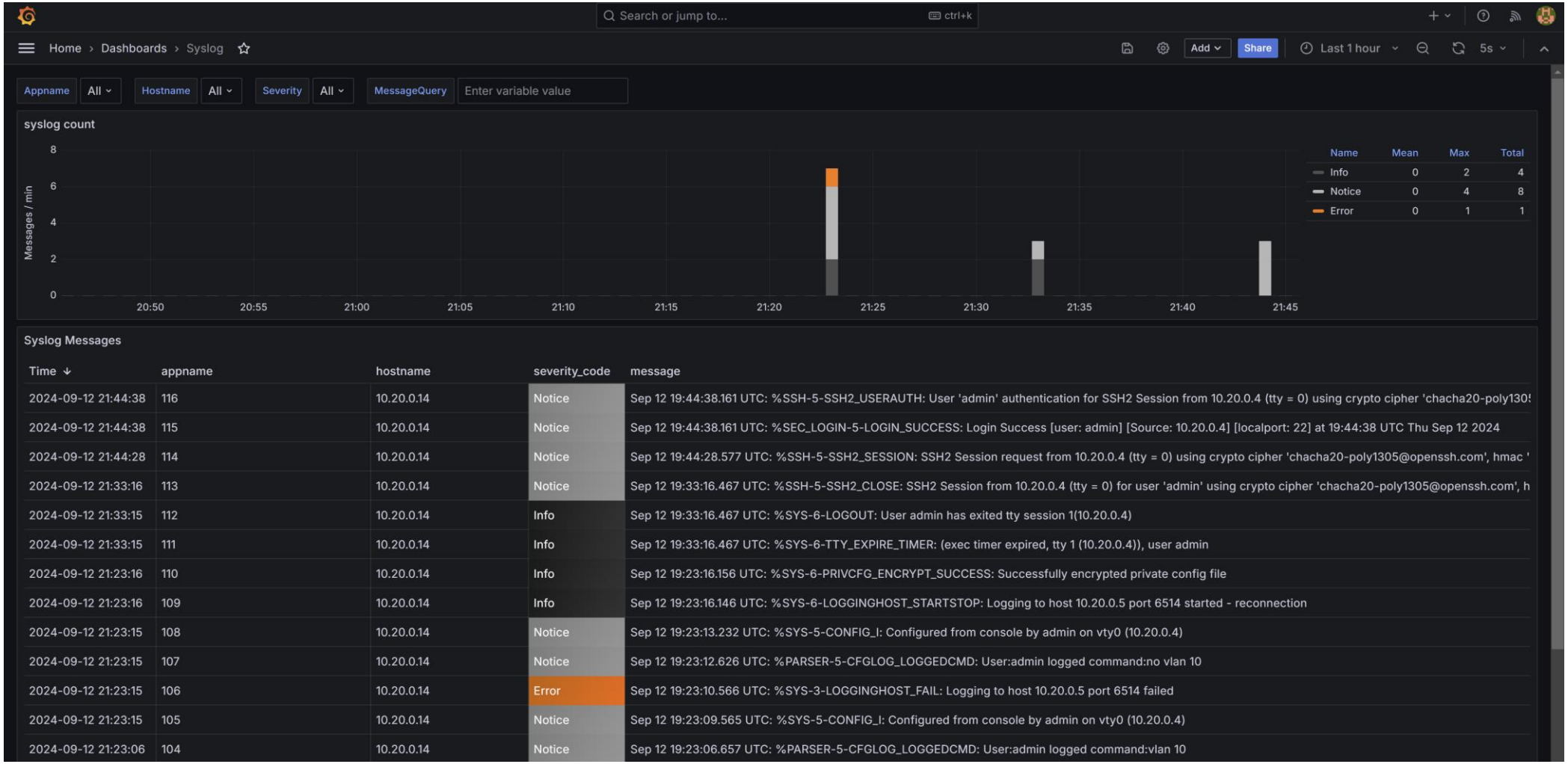


- Give it a name

- Import dashboard: 12433



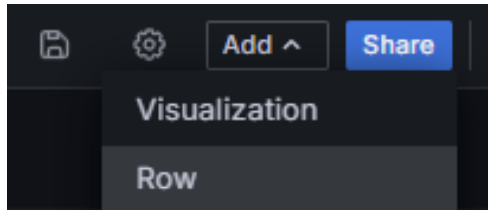
Lab Exercise #30: Grafana - Syslog

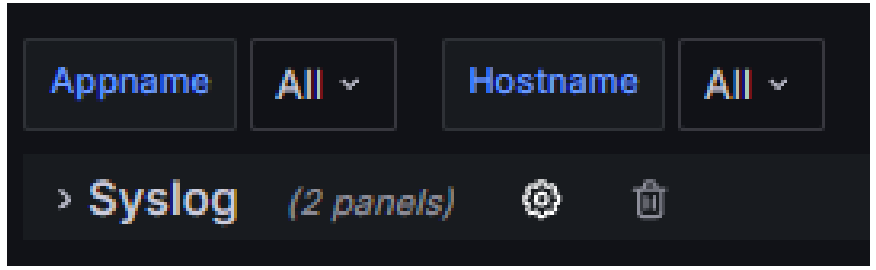


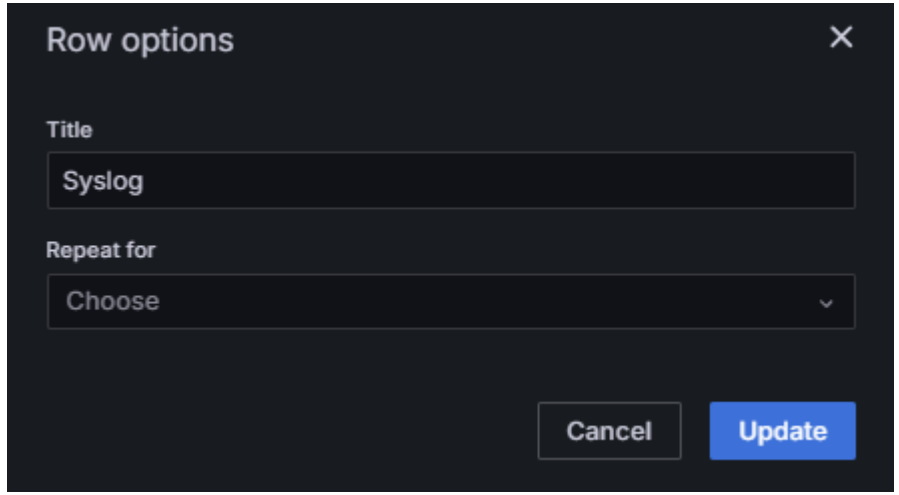
Lab Exercise #30: (Optional) Grafana – Syslog (row)

We can move the syslog panels in a drop-down, row. Useful is you want multiple panels in your Grafana dashboard, and you want a hide a section.

- Add a row from the top-menu
- Change the Title by selecting gear icon.
  - All panels below row will now be inside the row section







Lab Exercise #30: Grafana - Syslog

- Search for syslogs here

- All config-commands

Lab Exercise #31: Grafana – C9800 data

- Let us add some data from our 9800 WLC?

Do not want to do this lab? You can download the syslog and 9800 Dashboard here:
You will need to change the datasource variable in GUI or in json, but quicker than lab ☺

C9800 Stats + Syslog-1729418435270.json

Lab Exercise #31: Grafana – C9800 data

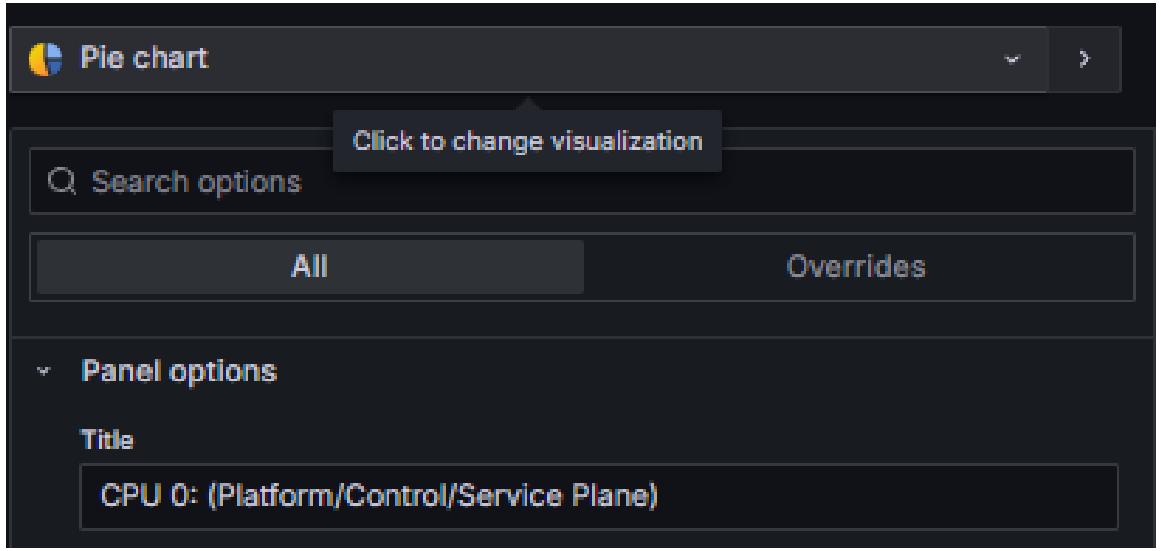
- Add the following telemetry data to your lab WLC
  - The lab WLC have these configured already.

```
telemetry ietf subscription 301
 encoding encode-kvgpb
 filter xpath /ios:native/version
 stream yang-push
 update-policy periodic 30000
 receiver ip address 192.168.10.7 57000 protocol grpc-tcp
!
!
telemetry ietf subscription 302
 encoding encode-kvgpb
 filter xpath /platform-sw-ios-xe-oper:cisco-platform-software/platform-sw-ios-xe-oper:control-
processes/platform-sw-ios-xe-oper:control-process/platform-sw-ios-xe-oper:per-core-stats
 stream yang-push
 update-policy periodic 5000
 receiver ip address 192.168.10.7 57000 protocol grpc-tcp
!
!
telemetry ietf subscription 303
 encoding encode-kvgpb
 filter xpath /platform-sw-ios-xe-oper:cisco-platform-software/platform-sw-ios-xe-oper:control-
processes/platform-sw-ios-xe-oper:control-process/platform-sw-ios-xe-oper:memory-stats
 update-policy periodic 5000
 receiver ip address 192.168.10.7 57000 protocol grpc-tcp
!
!
telemetry ietf subscription 304
 encoding encode-kvgpb
 filter xpath /device-hardware-xe-oper:device-hardware-data/device-hardware-xe-oper:device-
hardware/device-hardware-xe-oper:device-system-data
 update-policy periodic 10000
 receiver ip address 192.168.10.7 57000 protocol grpc-tcp
!
```

```
telemetry ietf subscription 305
 encoding encode-kvgpb
 filter xpath /device-hardware-xe-oper:device-hardware-data/device-hardware-xe-oper:device-
hardware/device-hardware-xe-oper:device-inventory
 update-policy periodic 30000
 receiver ip address 192.168.10.7 57000 protocol grpc-tcp
!
!
telemetry ietf subscription 306
 encoding encode-kvgpb
 filter xpath /wireless-ap-global-oper:ap-global-oper-data/wireless-ap-global-oper:ap-join-stats
 stream yang-push
 update-policy periodic 30000
 receiver ip address 192.168.10.7 57000 protocol grpc-tcp
!
!
telemetry ietf subscription 307
 encoding encode-kvgpb
 filter xpath /wireless-ap-global-oper:ap-global-oper-data/wireless-ap-global-oper:wlan-client-stats
 stream yang-push
 update-policy periodic 5000
 receiver ip address 192.168.10.7 57000 protocol grpc-tcp
!
!
```

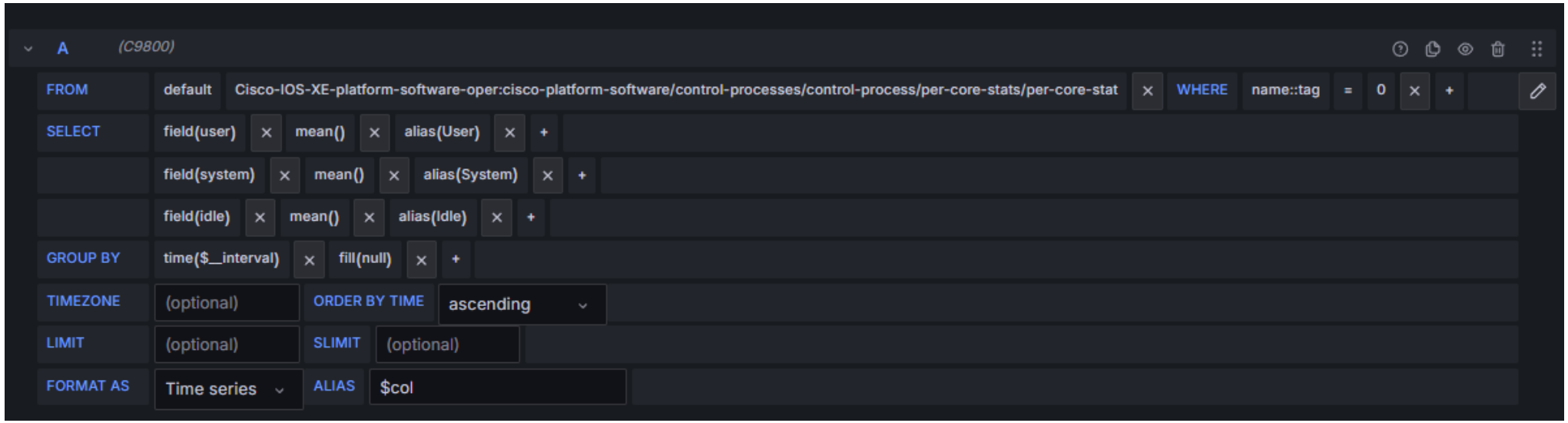
Lab Exercise #31: Grafana – C9800 data (CPU)

- Use your Syslog Dasbhoard and create a new visualization
- Select pie chart and name it:
  - CPU 0: (Platform/Control/Service Plane)



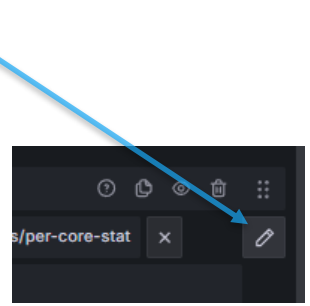
Lab Exercise #31: Grafana – C9800 data (CPU)

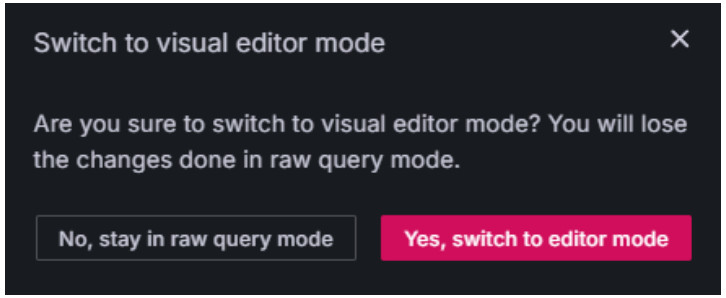
- Add the following query



- Alternative you can paste in the query raw, but that you cannot og back to editor mode.

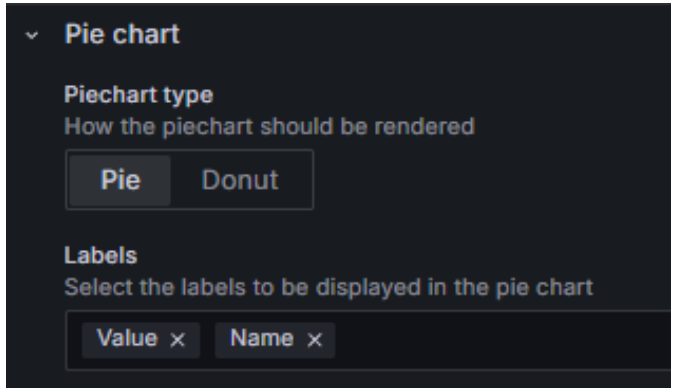
```
SELECT mean("user") AS "User", mean("system") AS "System", mean("idle") AS "Idle" FROM "Cisco-IOS-XE-platform-software-oper:cisco-platform-software/control-processes/control-process/per-core-stats/per-core-stat" WHERE ("name"::tag = '0') AND $timeFilter GROUP BY time($__interval) fill(null)
```



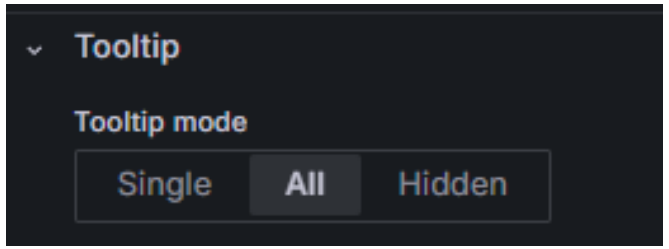


Lab Exercise #31: Grafana – C9800 data (CPU)

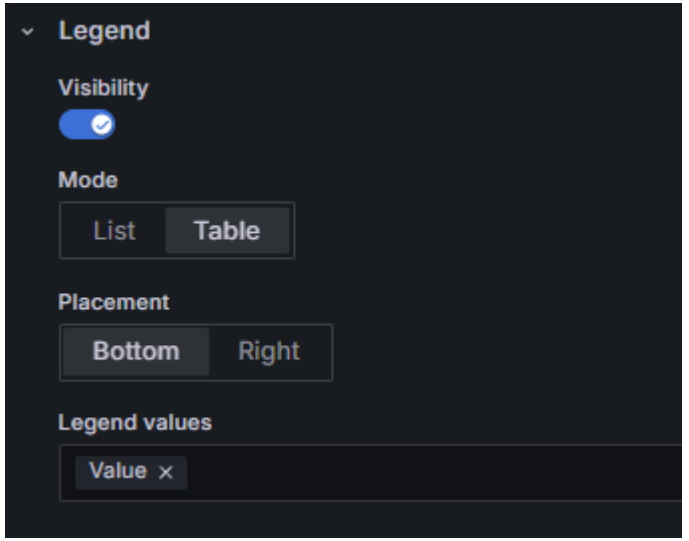
- Change the following panel options
  - Pie Chart Label: Value and name



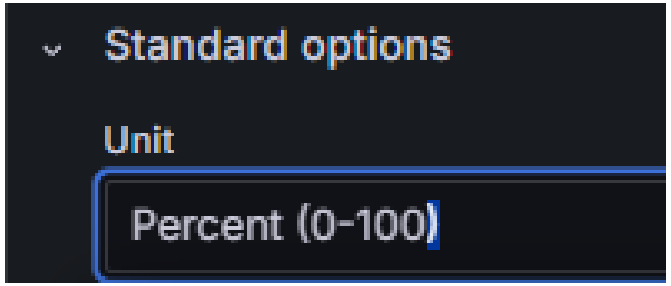
  - Tooltip is set to All



  - Legend: Table, Bottom, Value

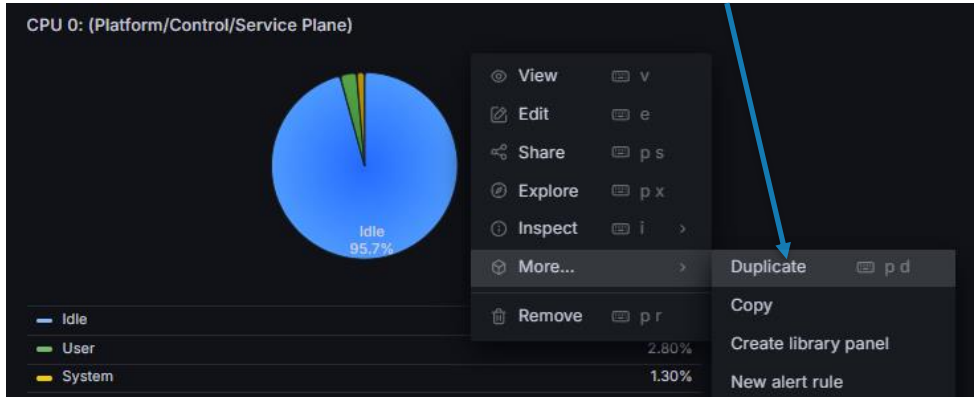


  - Standard Options: Percent

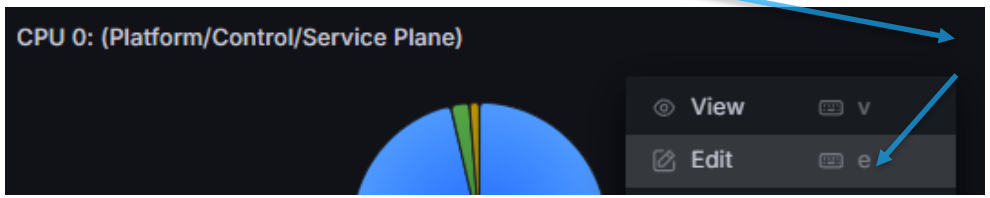


Lab Exercise #31: Grafana – C9800 data (CPU)

- Duplicate the panel you just made

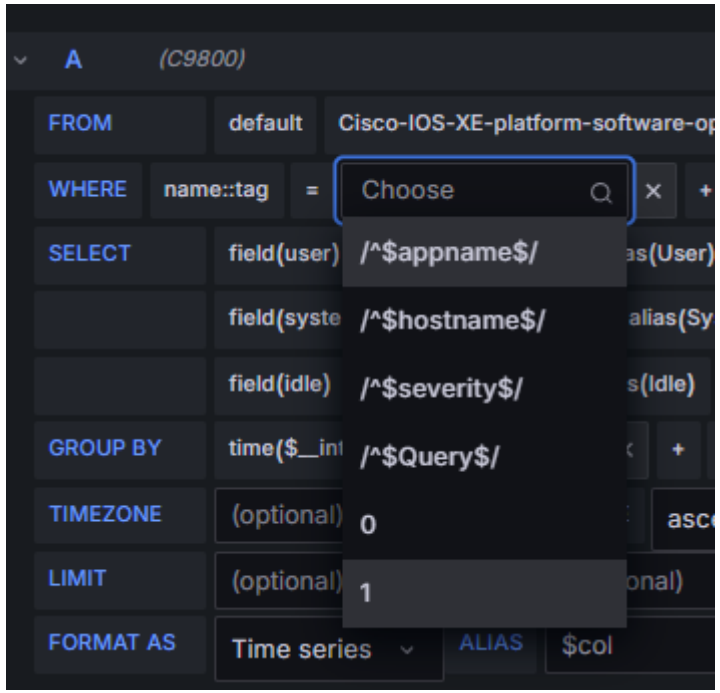


- Edit the duplicated panel

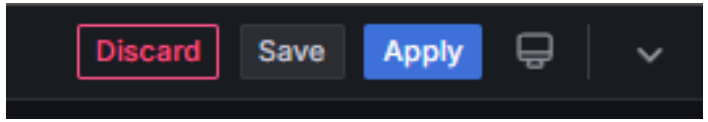


- Rename it to: CPU 1: (Data Plane)

- Change the name::tag in the query to 1

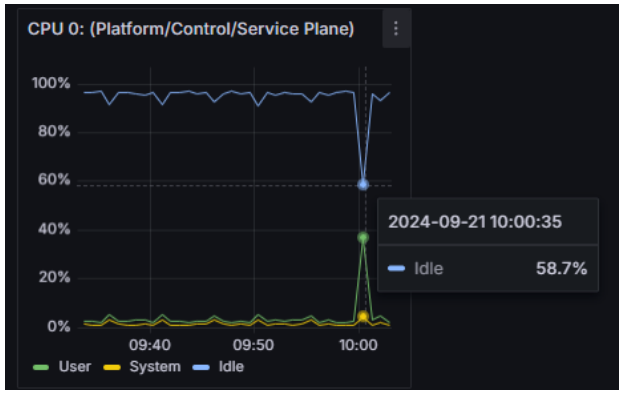


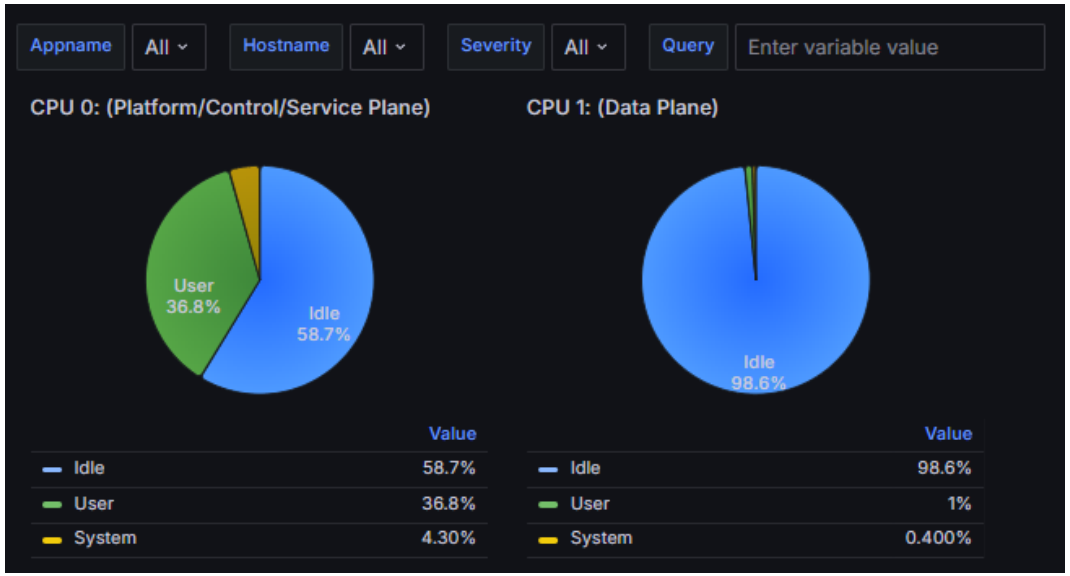
- Save and Apply

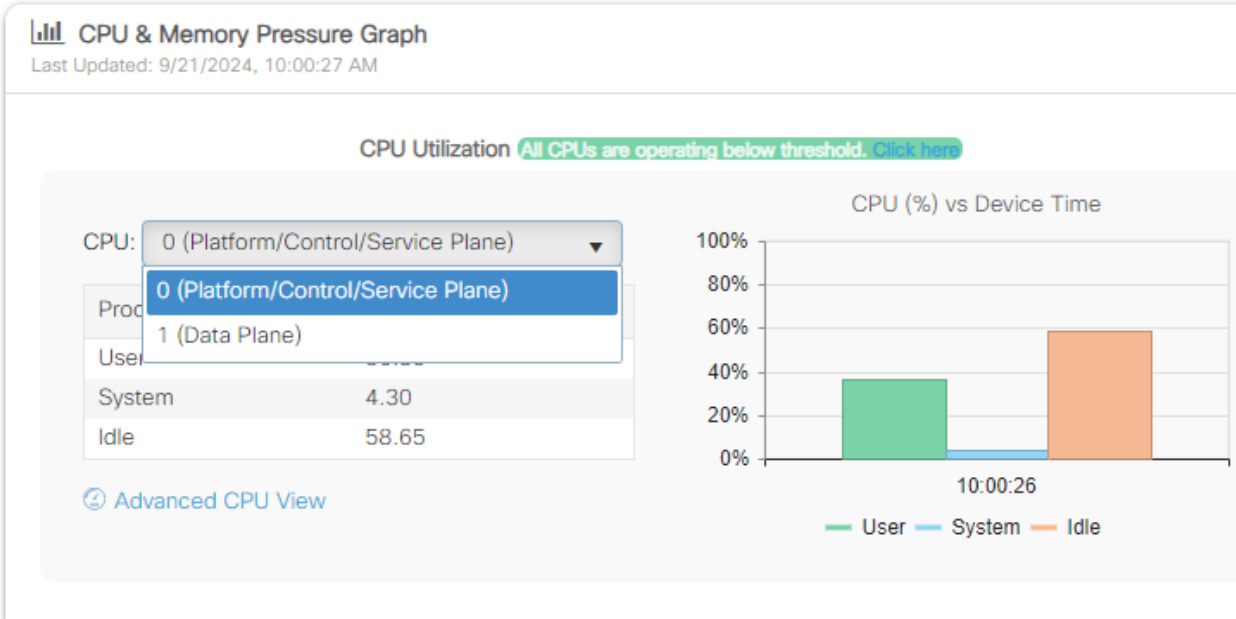


Lab Exercise #31: Grafana – C9800 data (CPU)

- Resize them to be a bit smaller and place them to the left.
  - Seen these before? ;)
- **(Optional)** These can also be made to a time-graph
  - See the CPU spike I had for a few seconds

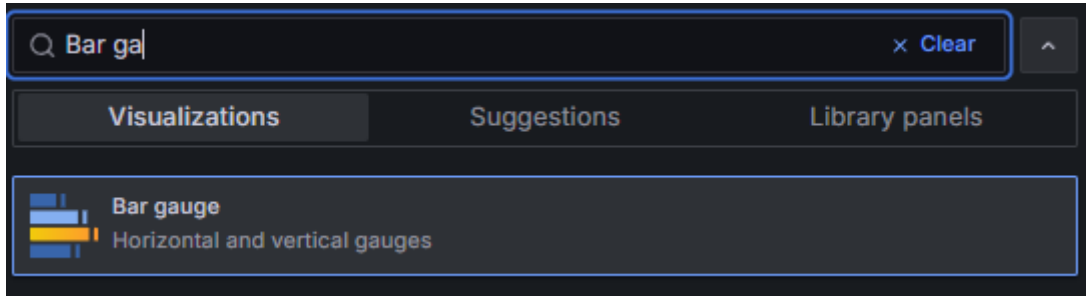


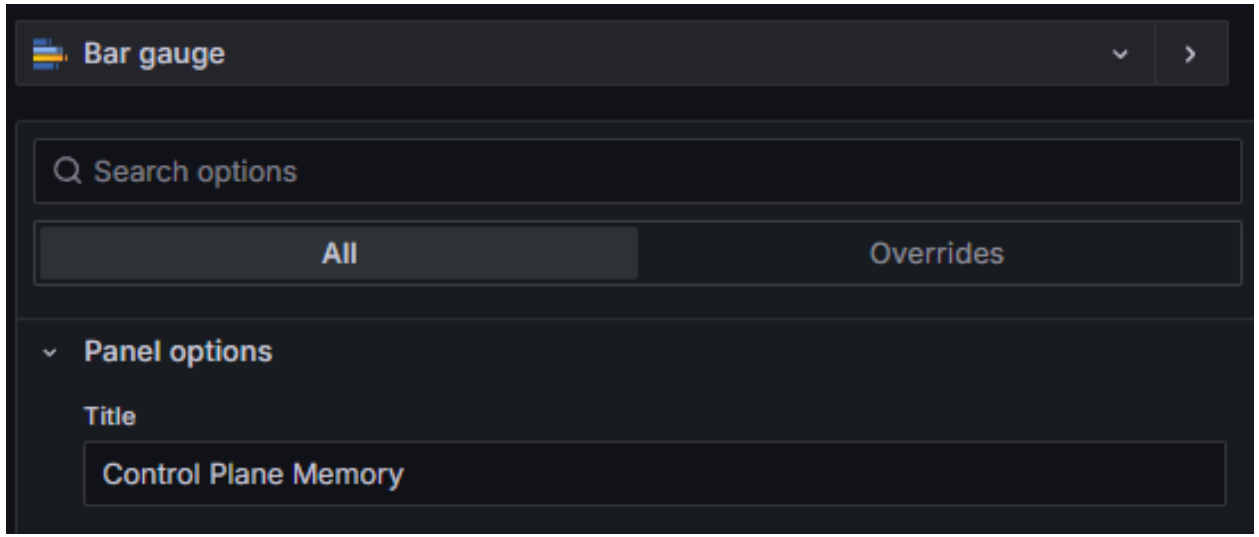




Lab Exercise #31: Grafana – C9800 data (Memory)

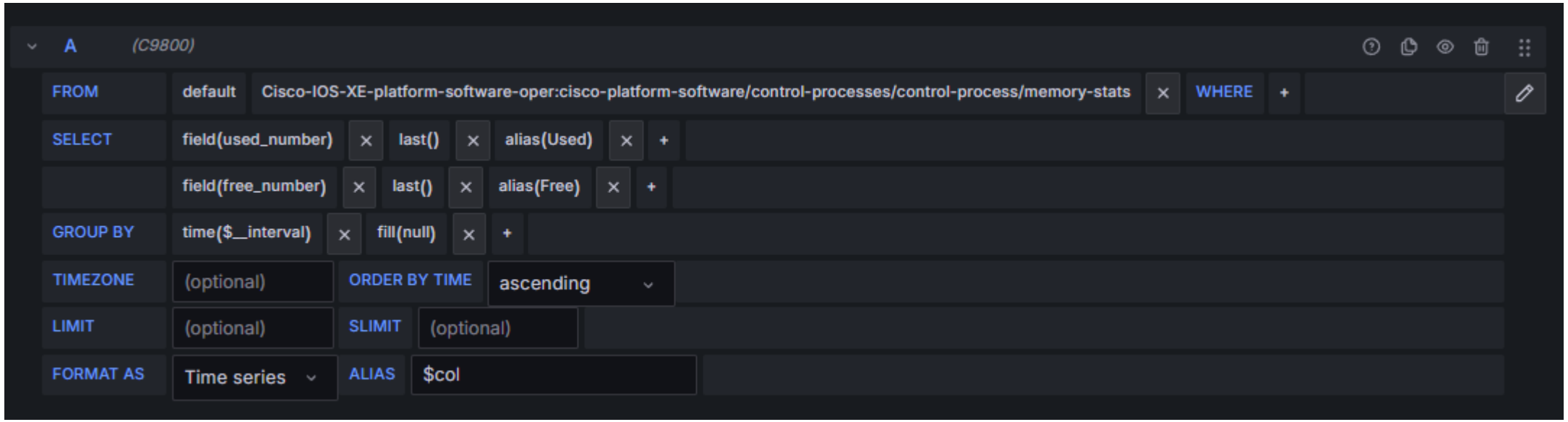
- Add a new panel, this time Bar Gauge (or Pie Chart)
- Give the panel a name
  - Control Plane Memory





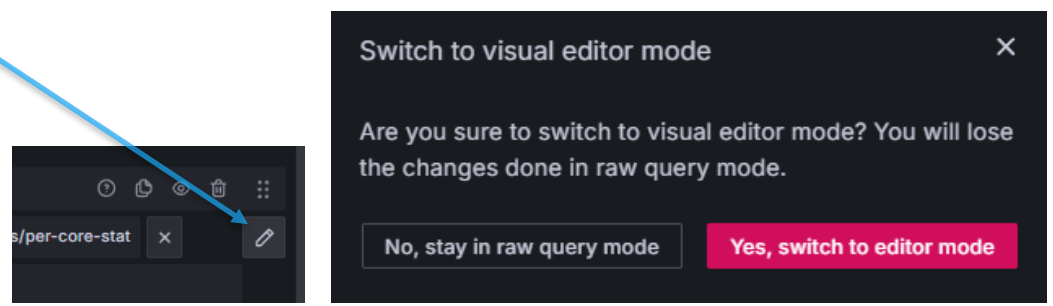
Lab Exercise #31: Grafana – C9800 data (Memory)

- Add the following query



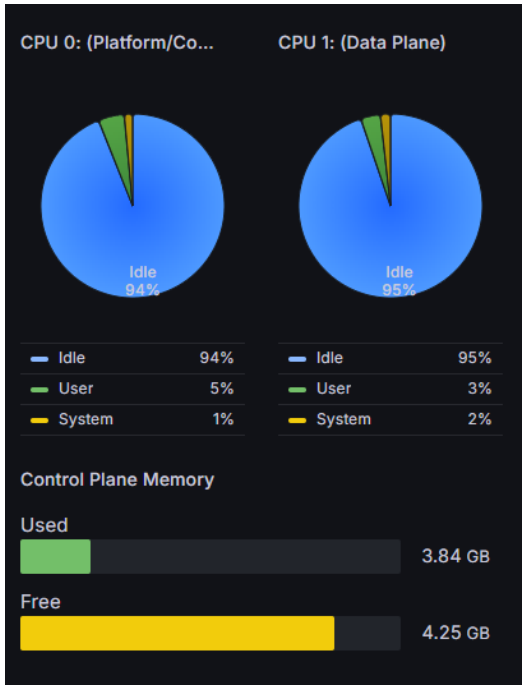
- Alternative you can paste in the query raw, but that you cannot og back to editor mode.

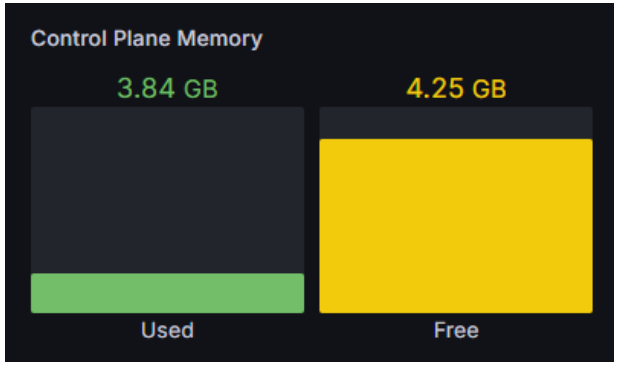
```
SELECT last("used_number") AS "Used",
last("free_number") AS "Free" FROM "Cisco-IOS-XE-
platform-software-oper:cisco-platform-software/control-
processes/control-process/memory-stats" WHERE
$timeFilter GROUP BY time($__interval) fill(null)
```



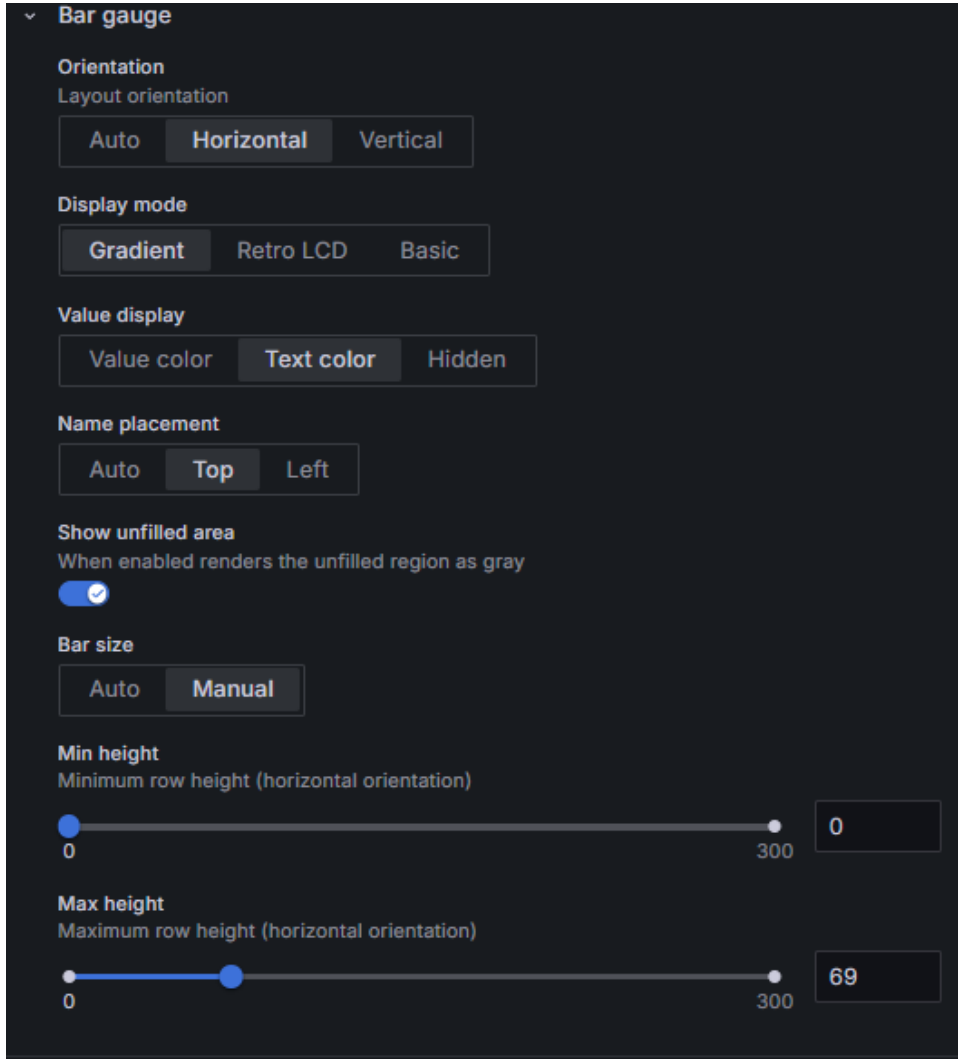
Lab Exercise #31: Grafana – C9800 data (Memory)

- Change some settings in the Bar Gauge panel options
  - Selelect the Horizontal orientation (or vertical) whatever you prefer.
  - Set Bar Size to Manual and change it to fit inside your panel.
  - Place this panel below the CPU panel.



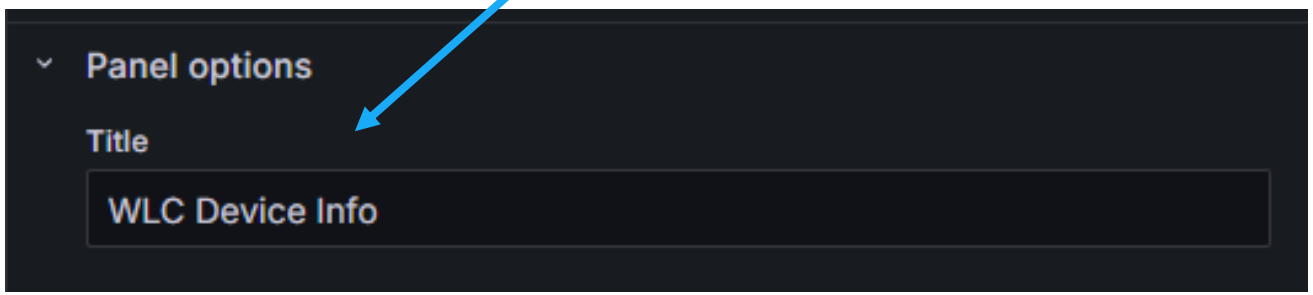


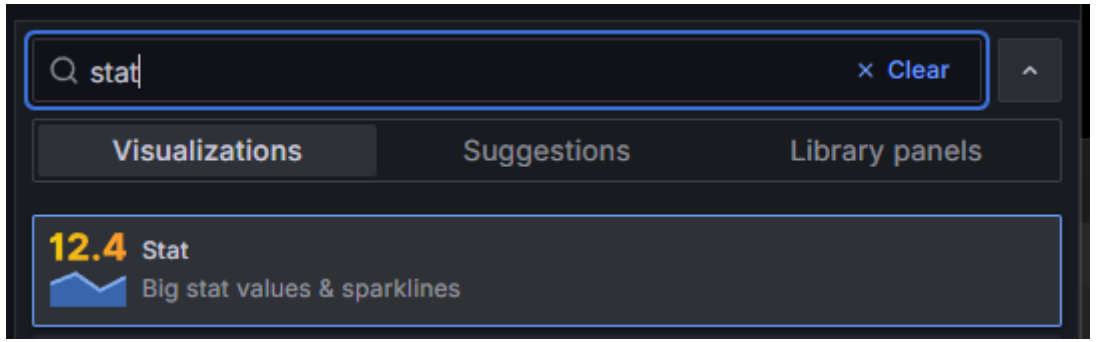
Vertical orientation



Lab Exercise #31: Grafana – C9800 data (Memory)

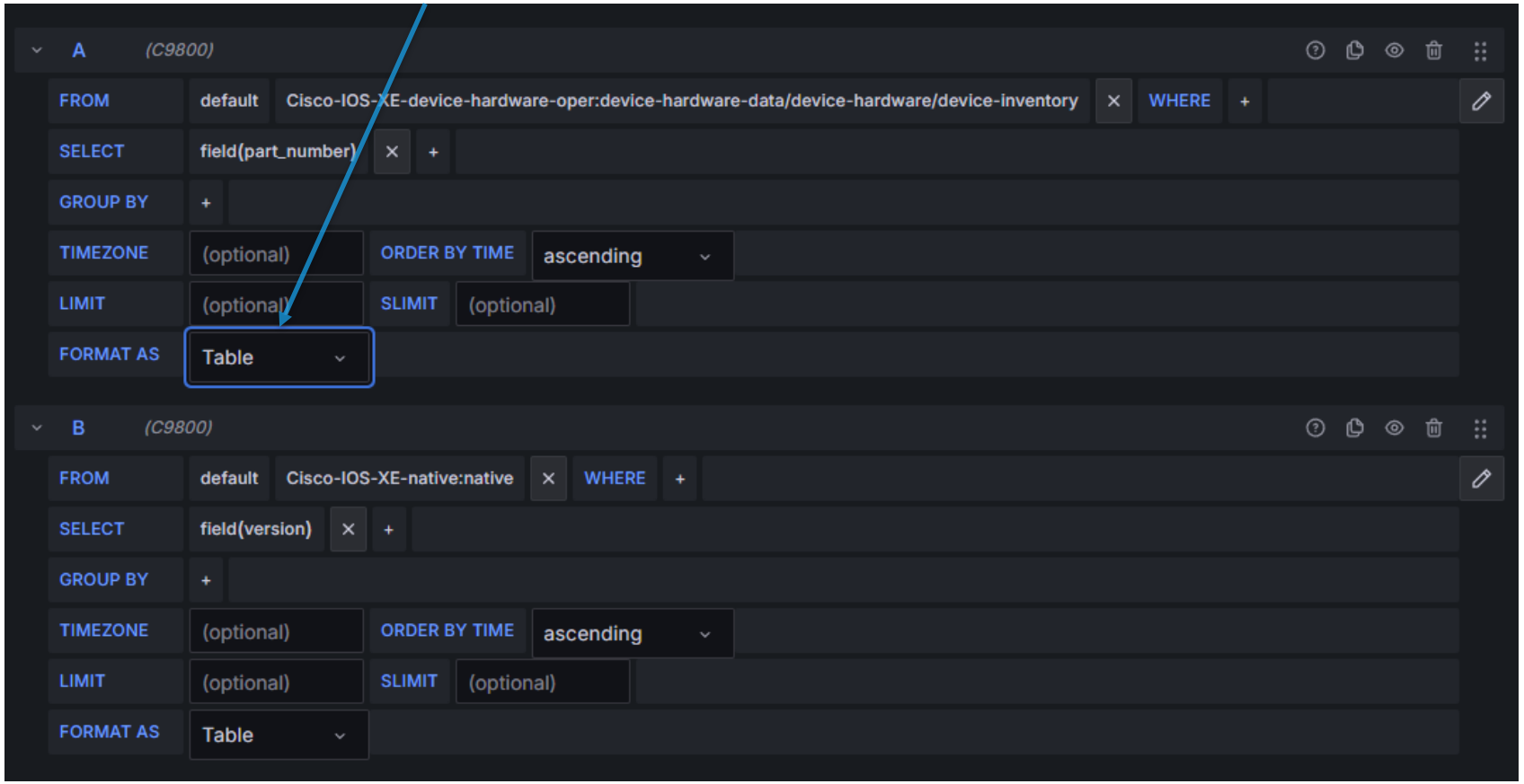
- Add a new panel for Device Info.
- Name the first panel «WLC Device Info»





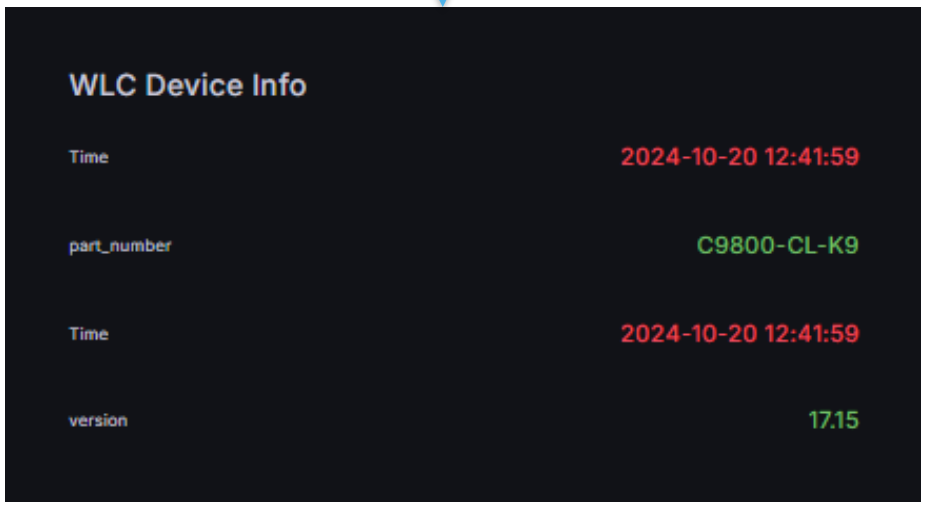
Lab Exercise #31: Grafana – C9800 data (Memory)

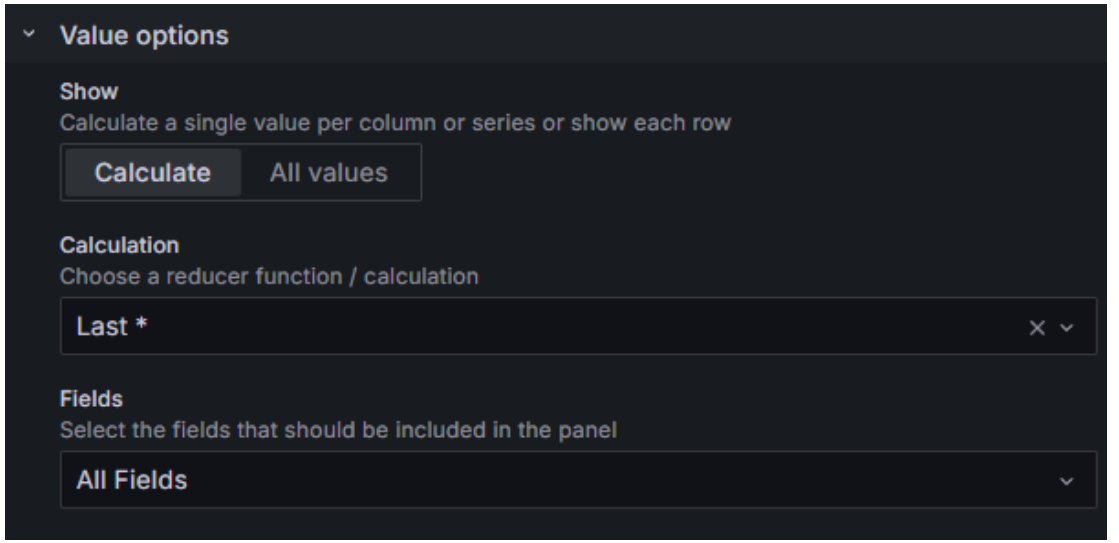
- Add the following queries
- Select, format as Table.

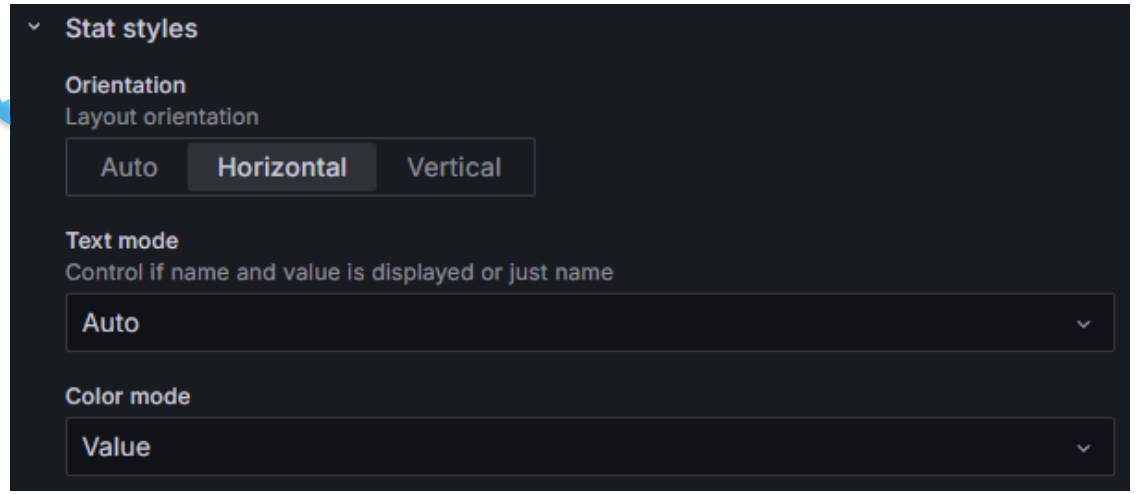


Lab Exercise #31: Grafana – C9800 data (Memory)

- Change the Value option to **Last*** and Fields to **All Fields.**
- Change the Stat styles to Horizontal
- Your panel should now look like this.



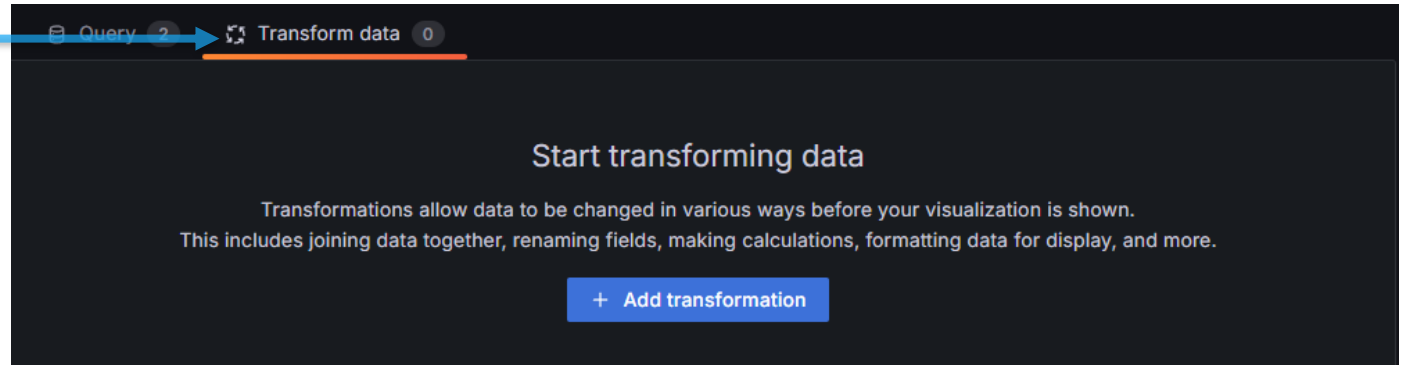


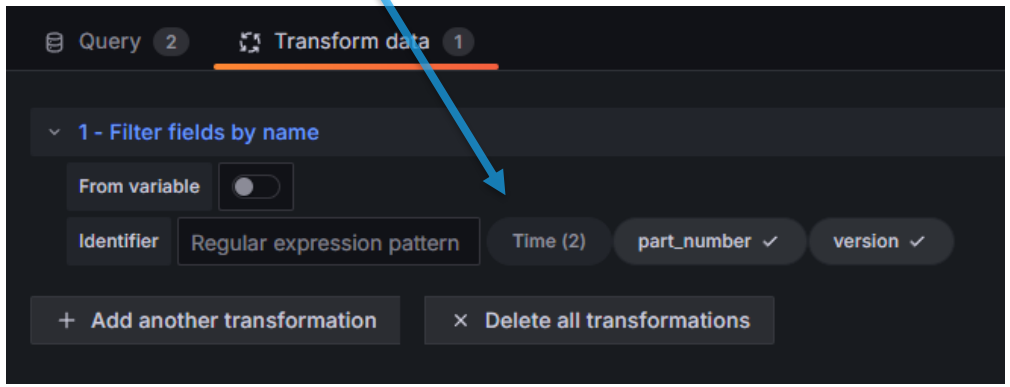


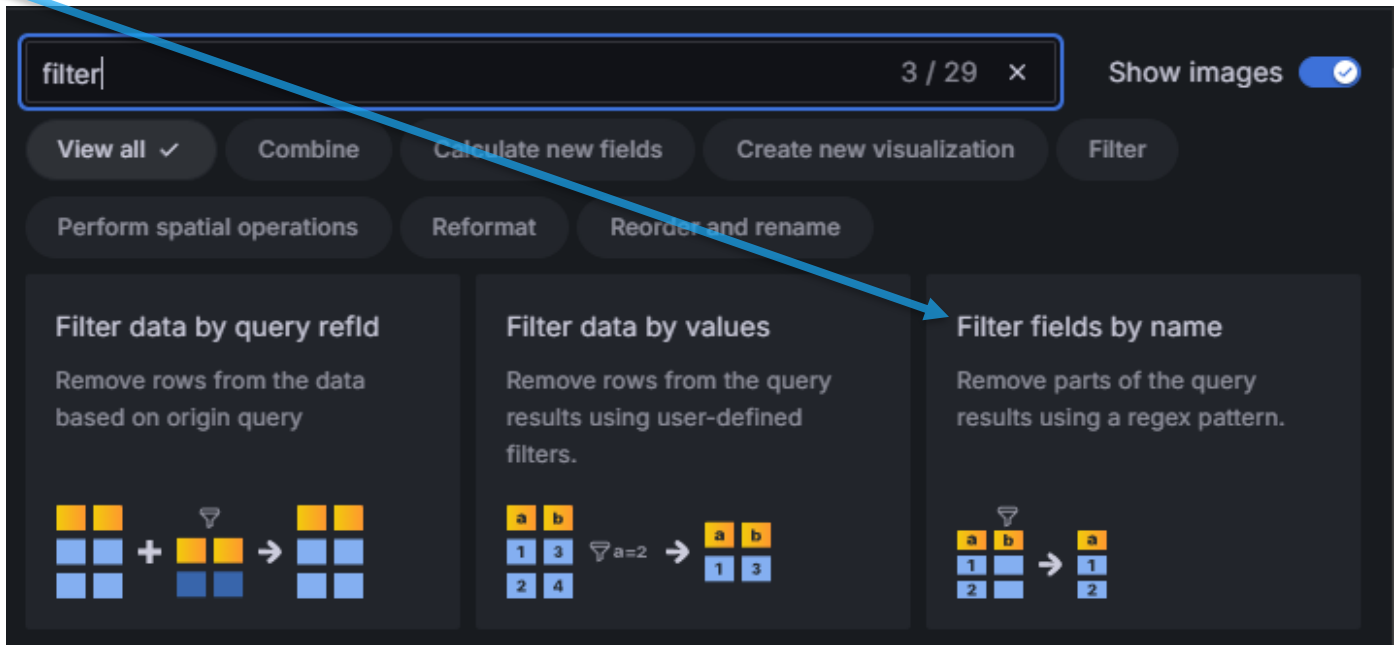
On the next slide we will remove time variabel

Lab Exercise #31: Grafana – C9800 data (Memory)

- Add a transformation
- Search for filter, and select **«Filter fields by name»**
- **Remove time**

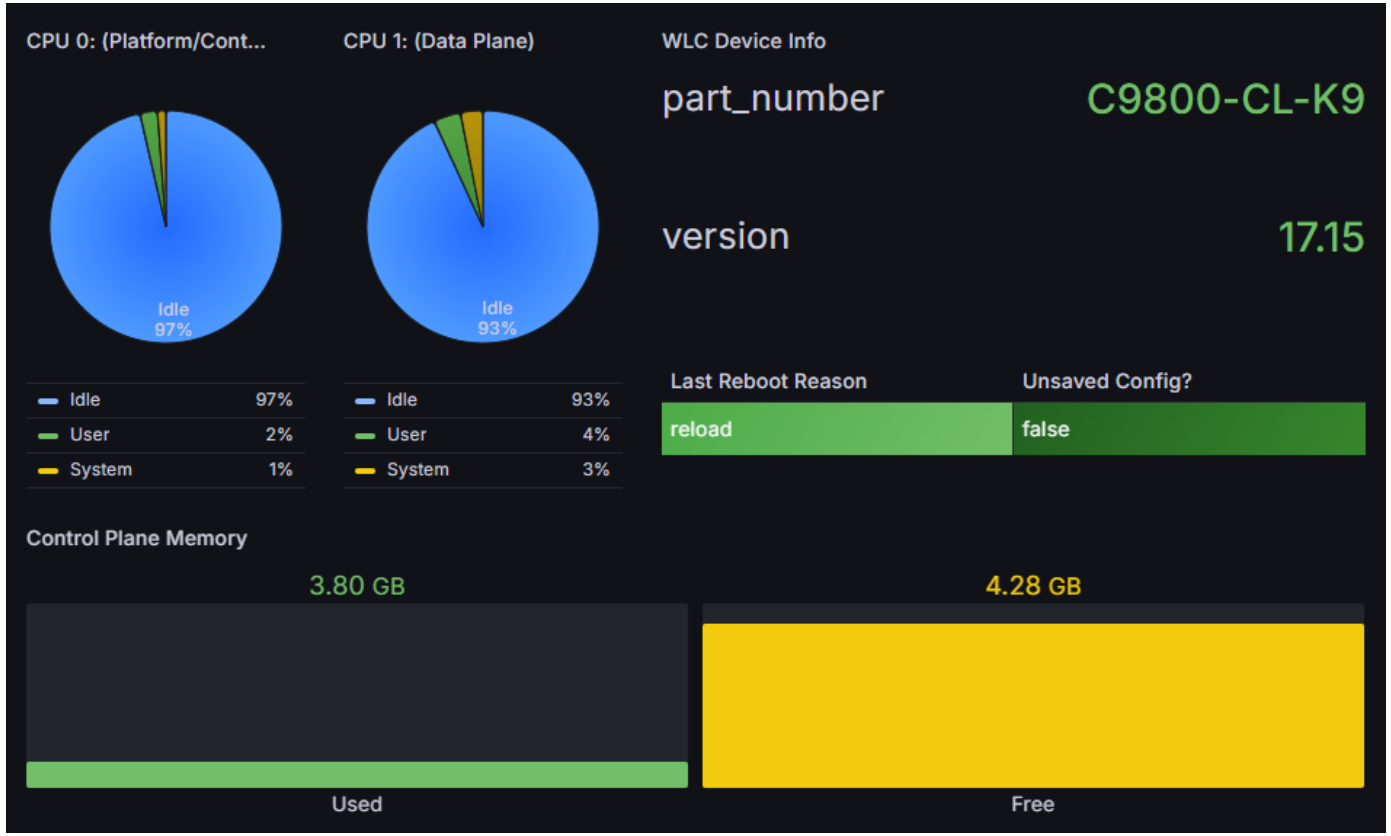






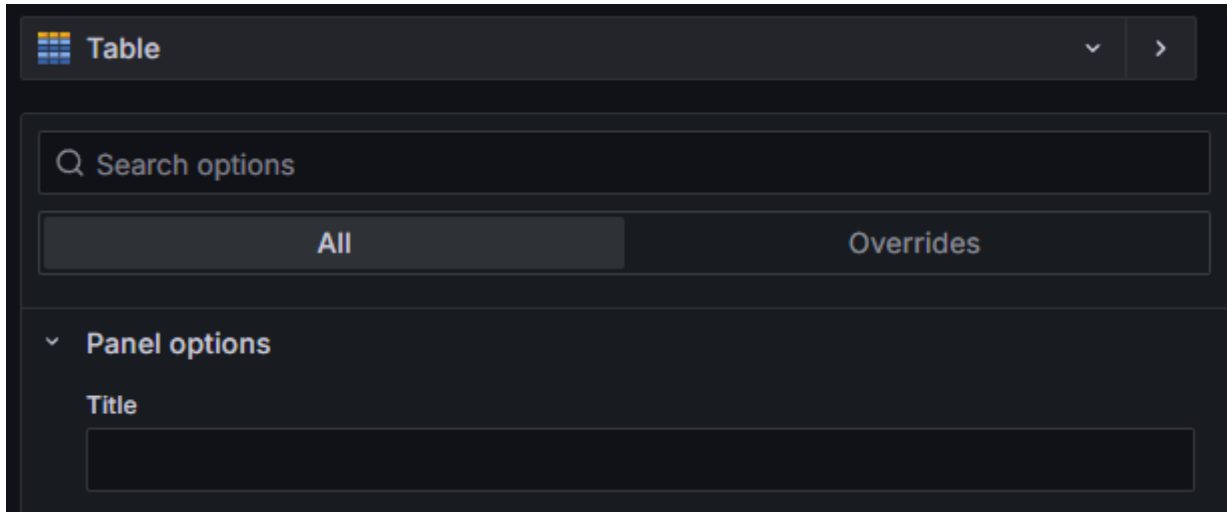
Lab Exercise #31: Grafana – C9800 data (Memory)

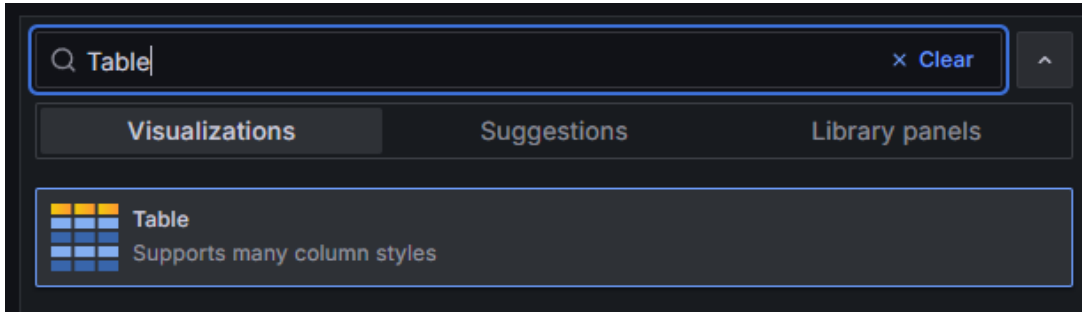
- Add the panel next to the CPU panel, or wherever you like to place it ☺
- Make the Control Plane Memory wider.
  - When we add more device info variables, we can shrink it again, this is what makes Grafana so useful.



Lab Exercise #31: Grafana – C9800 data (reload, config)

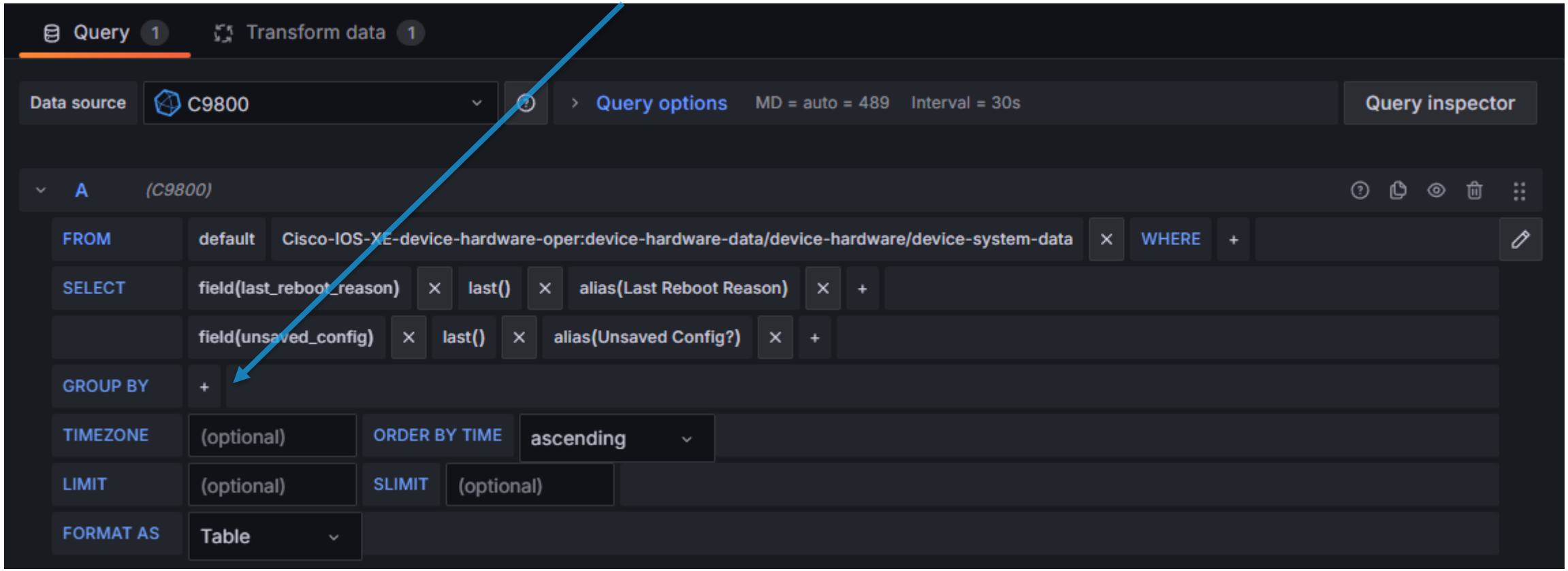
- Add a new panel for Last Reboot Reason and Unsaved Config.
  - Select tabel panel.
- Do not add any name for this panel.





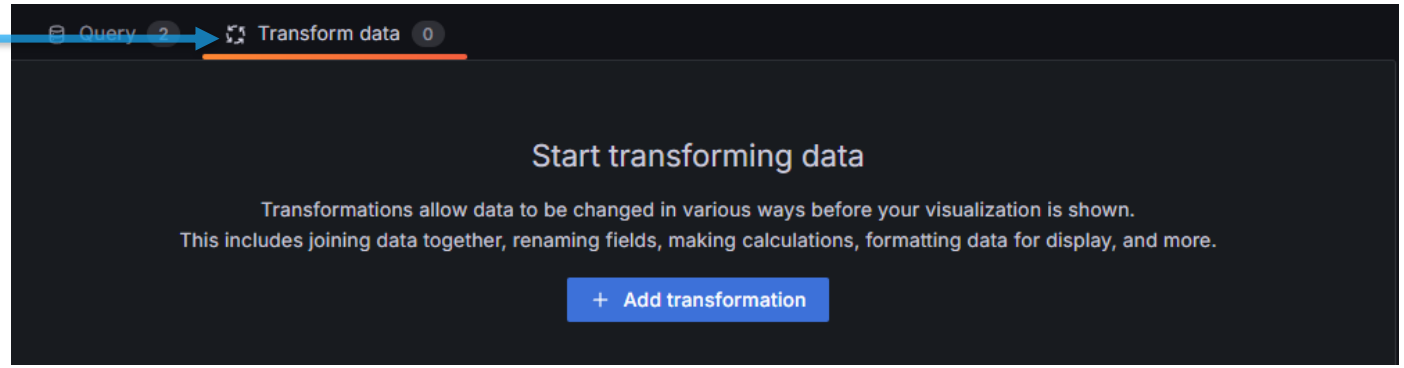
Lab Exercise #31: Grafana – C9800 data (reload, config)

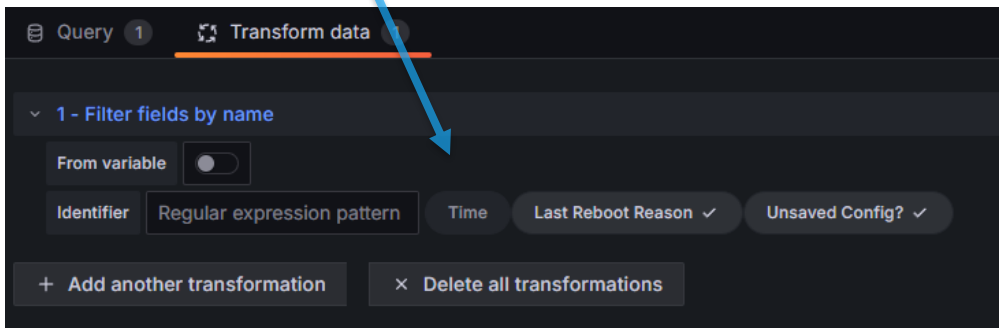
- Add the following queries
- Select, format as **Table** and **remove Group By**.

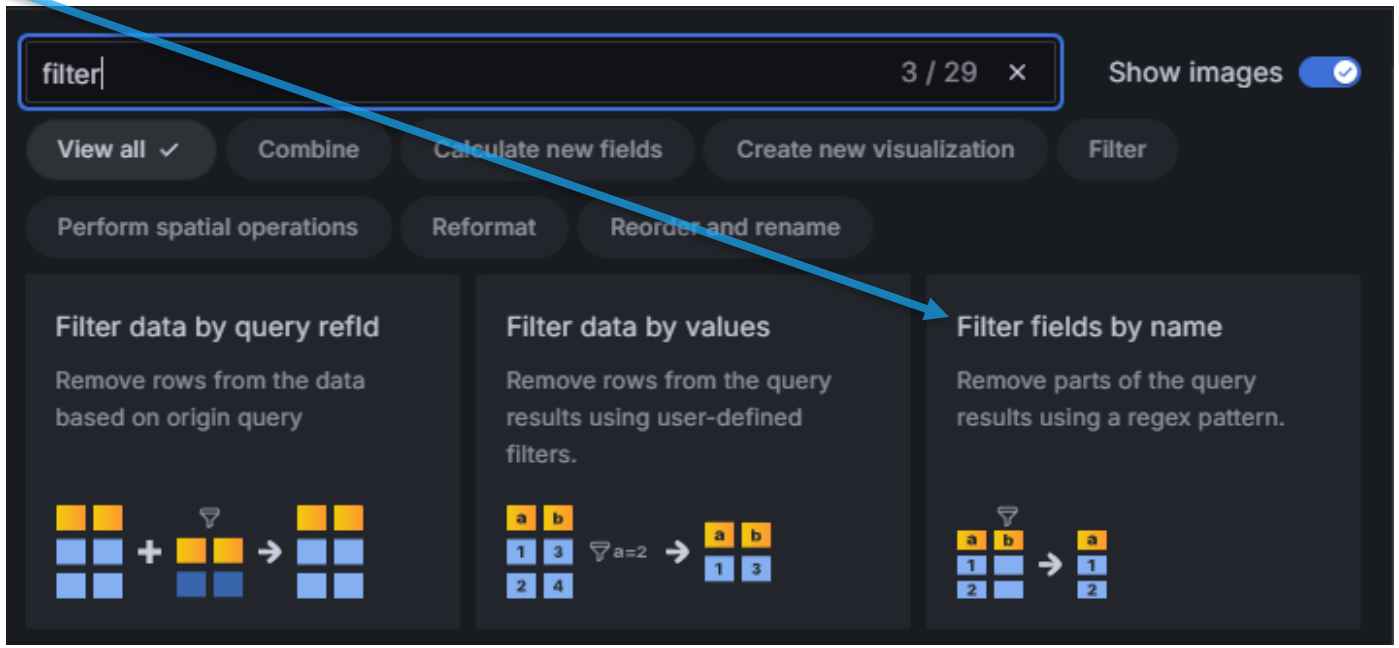


Lab Exercise #31: Grafana – C9800 data (reload, config)

- Add a transformation
- Search for filter, and select **«Filter fields by name»**
- **Remove time**

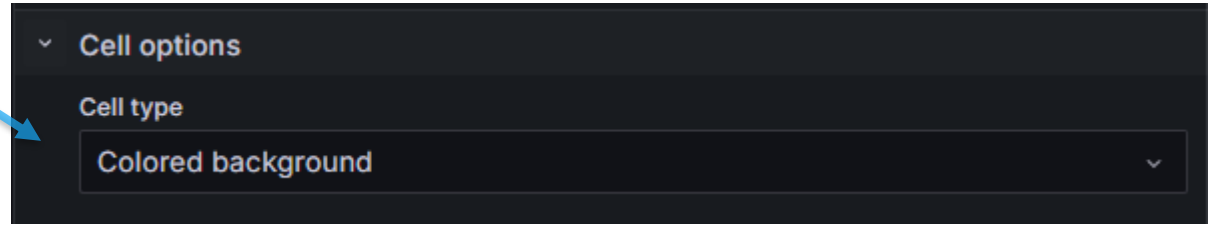


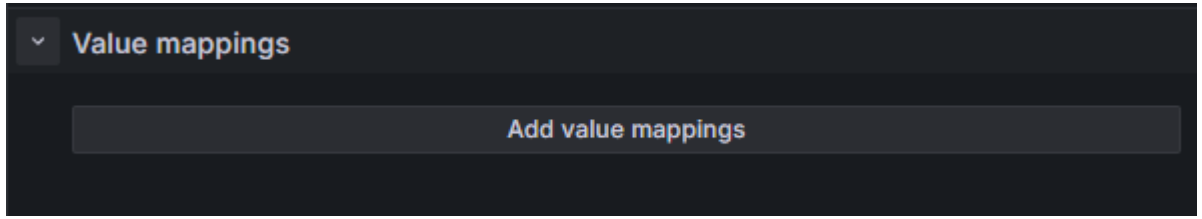




Lab Exercise #31: Grafana – C9800 data (reload, config)

- Change the Cell option to Colored Background
- Grafana will select «false» as red, but false here is positive since the config is saved ☺

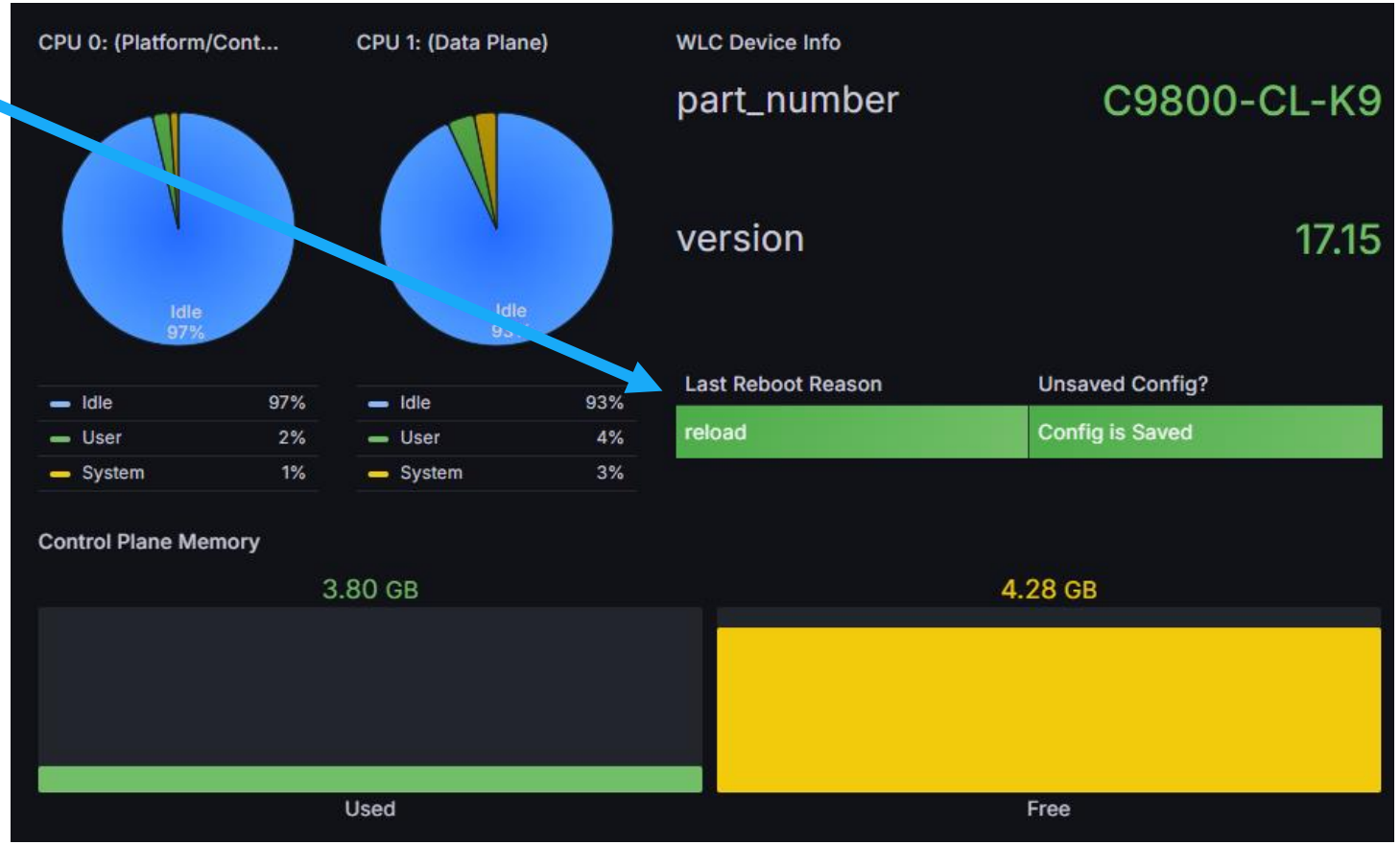




- Change false to green and true to red.
  - Add a two Value mappings
  - False to green (config is saved)
  - Ture to red (config is not saved)
  - (Optional) change the display text☺

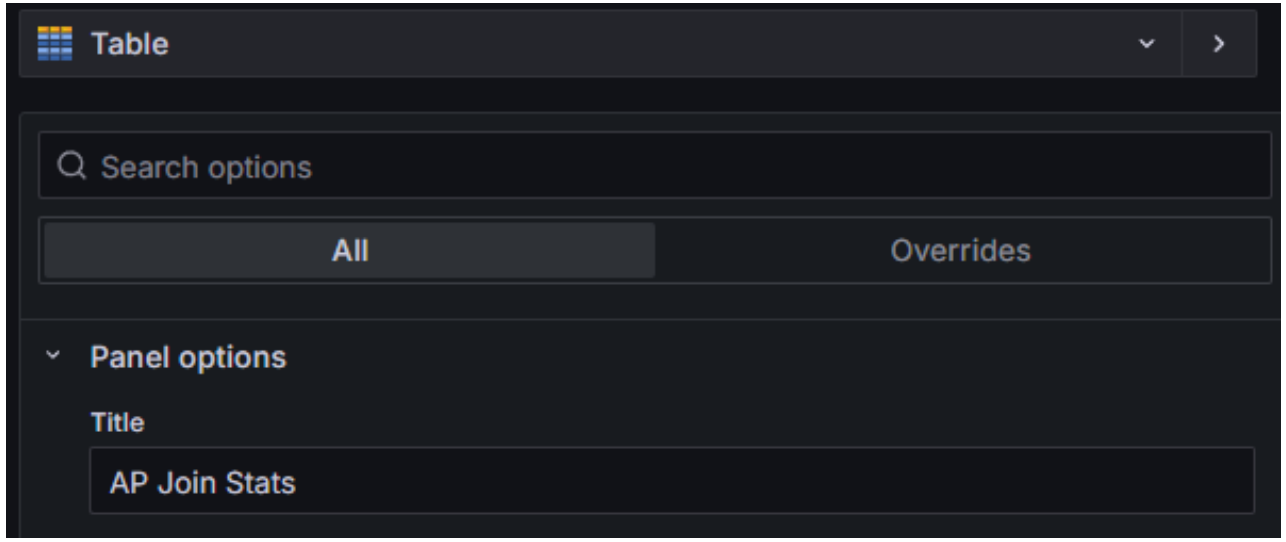
Lab Exercise #31: Grafana – C9800 data (reload, config)

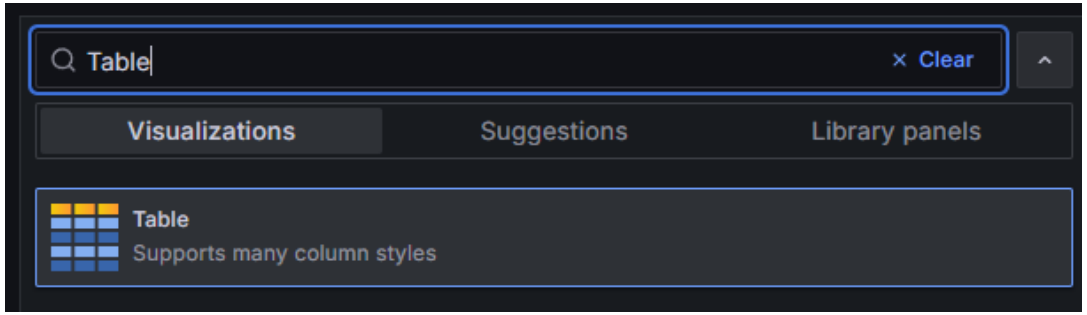
- Add this panel below like this



Lab Exercise #31: Grafana – C9800 data (AP Join Stats)

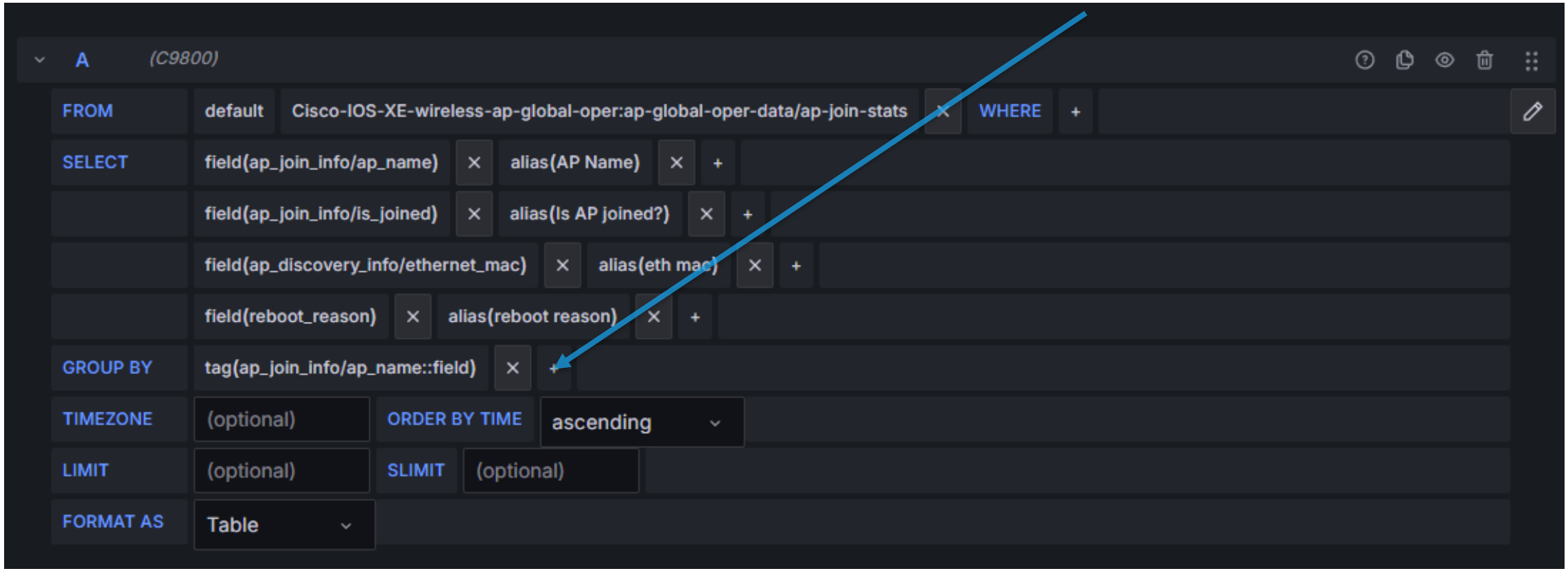
- Add a new panel for Last Reboot Reason and Unsaved Config.
  - Select tabel panel.
- Name the panel **AP Join Stats.**





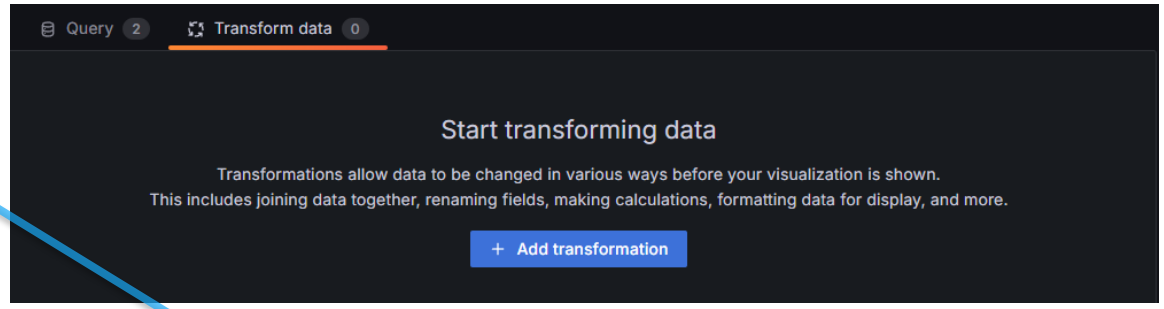
Lab Exercise #31: Grafana – C9800 data (AP Join Stats)

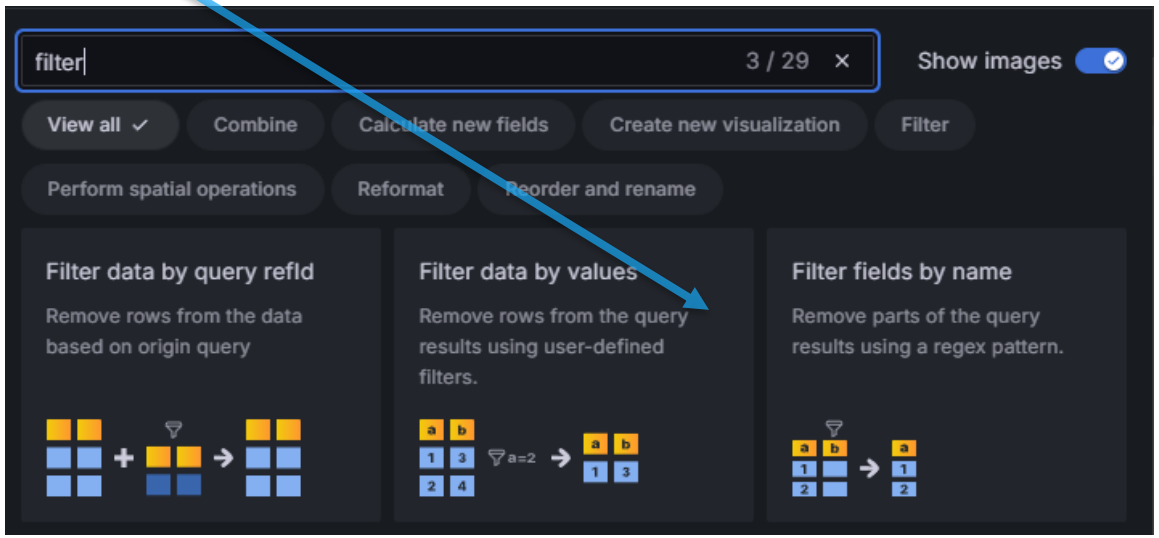
- Add the following queries
- Select, format as **Table** and Group by **tag(ap_join_info/ap_name::field)**



Lab Exercise #31: Grafana – C9800 data (AP Join Stats)

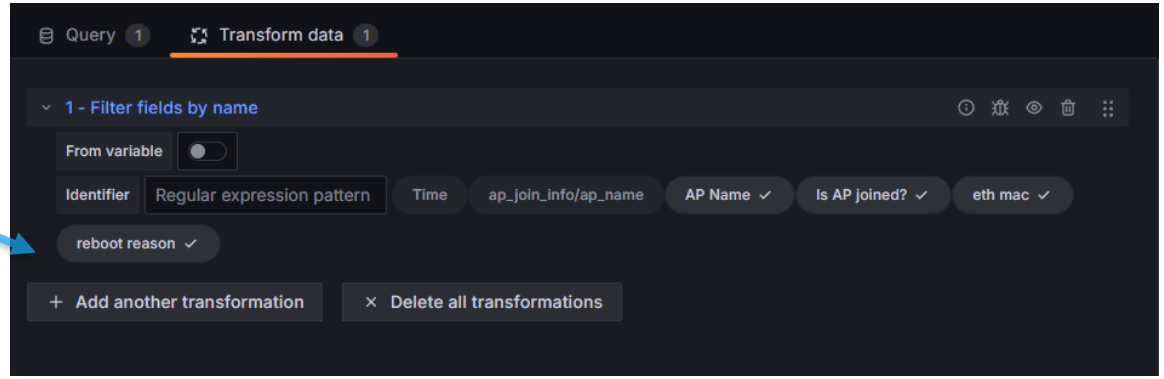
- Add a one transformations
- Search for filter, and select **«Filter fields by name»**
- Remove time Time and ap_join-info/ap_name

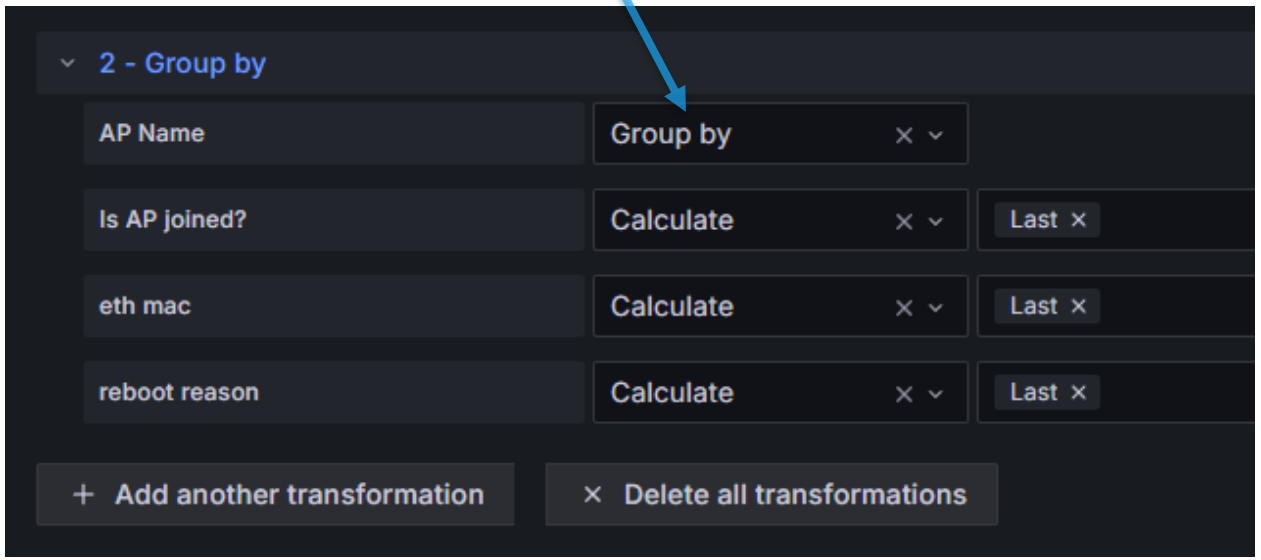


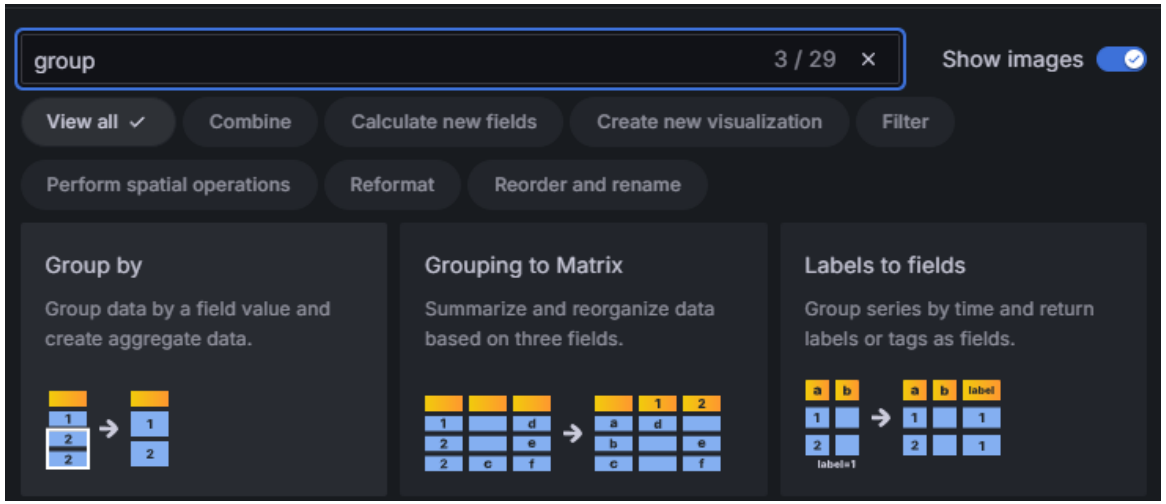


Lab Exercise #31: Grafana – C9800 data (AP Join Stats)

- Add a second transformations
- Search for filter, and select **«Group By»**
  - AP Name = Group By
  - All others = Last

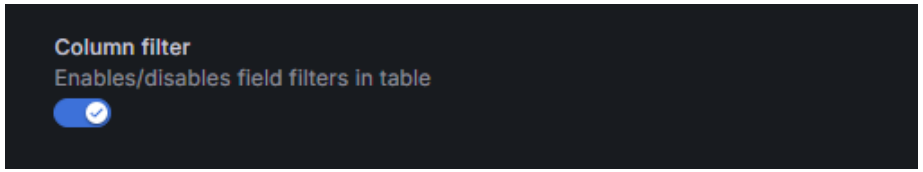


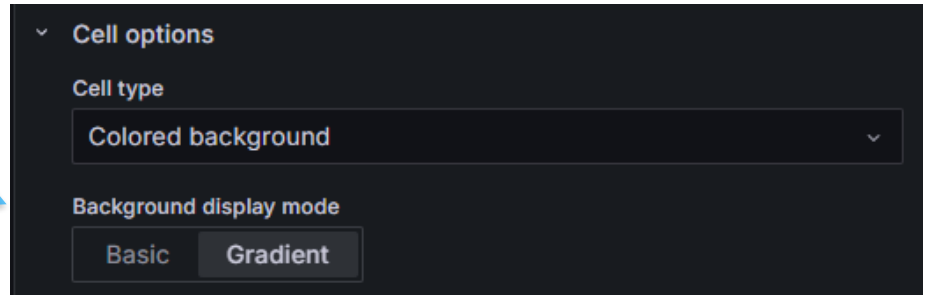


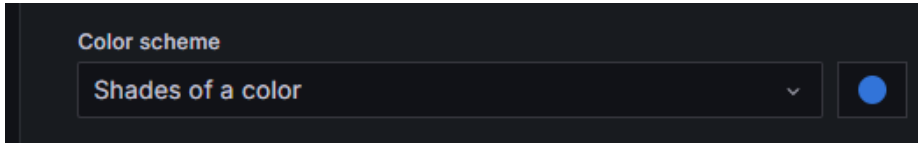


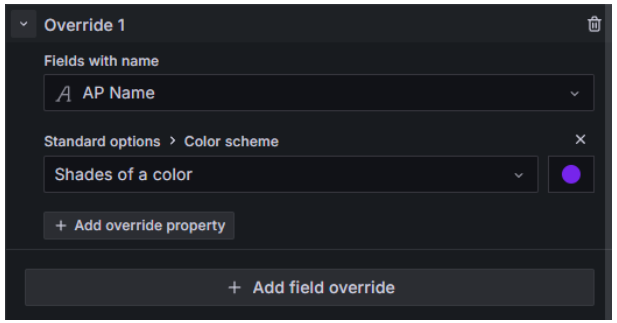
Lab Exercise #31: Grafana – C9800 data (AP Join Stats)

- Enable Table Column filter
  - Under **table** in panel options
- Under **Cell Options**, select Colored Background
- Under **Standard Options**, select Shades of color
  - Select a color that you like ☺
- **(Optional)** Make an override (at the bottom or top of panel) to change color of variables
  - Filed with name (AP Name)
  - Change color scheme







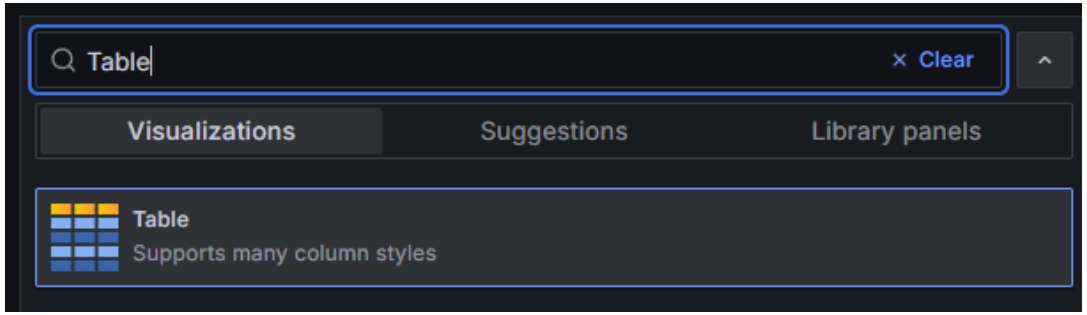


Lab Exercise #31: Grafana – C9800 data (AP Join Stats)

- Add this panel to the right of all your other panels

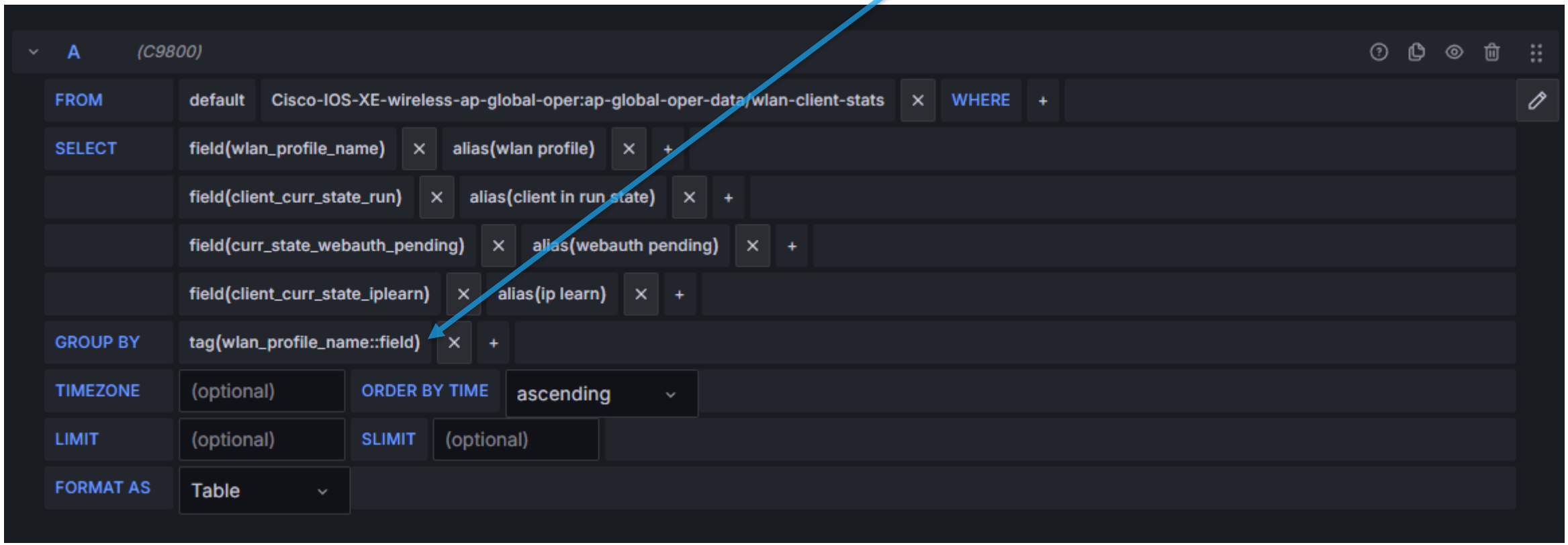
Lab Exercise #31: Grafana – C9800 data (WLAN Stats)

- Add a new panel for Last Reboot Reason and Unsaved Config.
  - Select tabel panel.
- Name the panel **WLAN Stats**



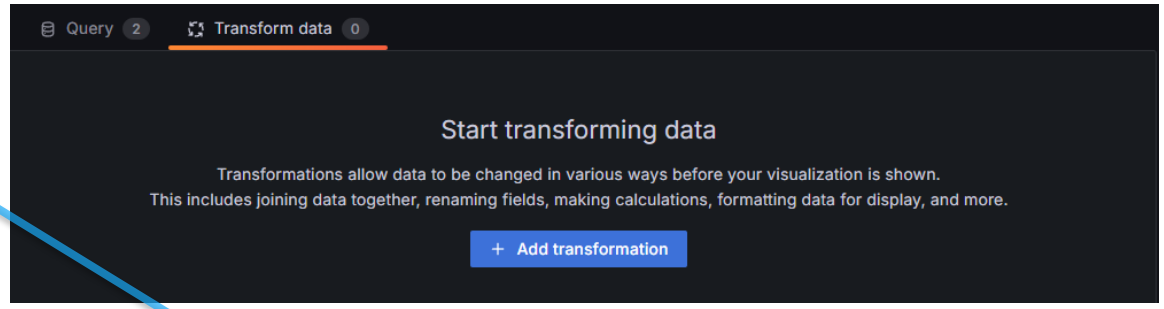
Lab Exercise #31: Grafana – C9800 data (WLAN Stats)

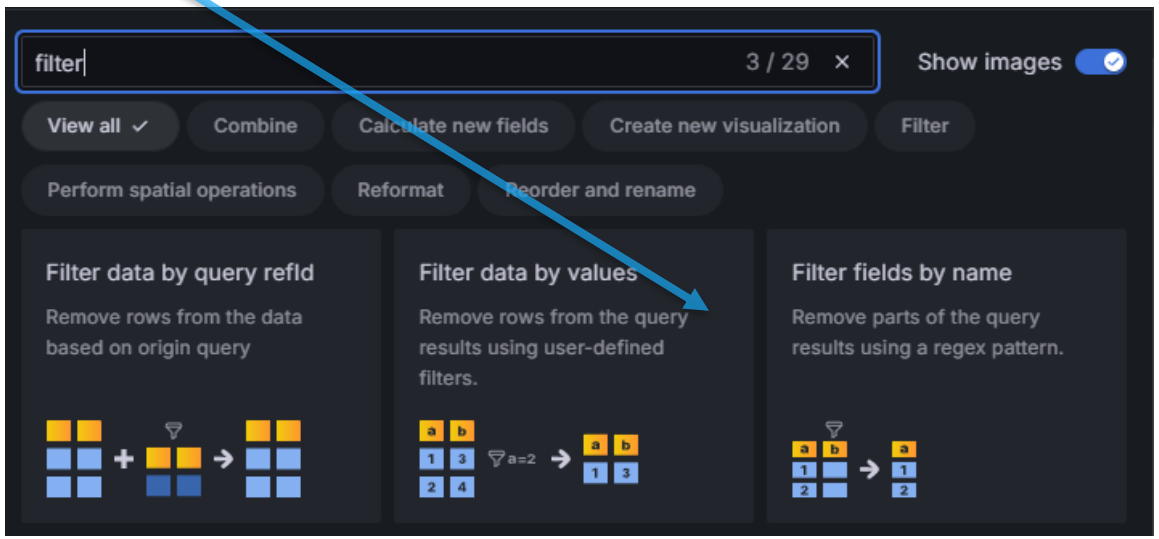
- Add the following queries
- Select, format as **Table** and Group by **tag(wlan_profile_name::field)**

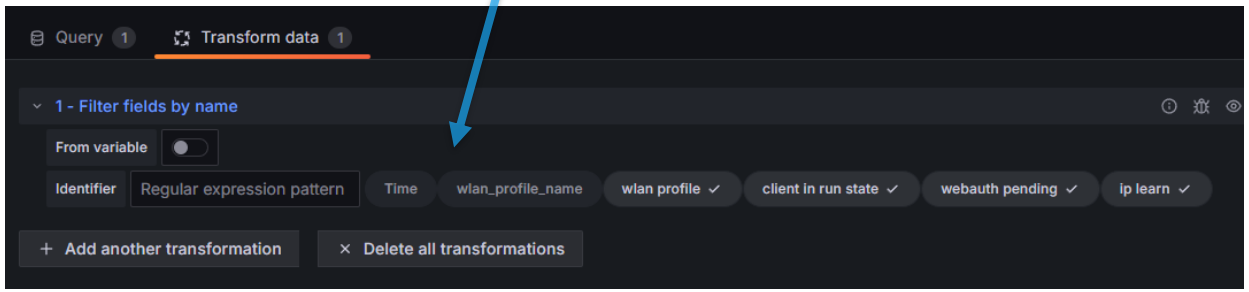


Lab Exercise #31: Grafana – C9800 data (WLAN Stats)

- Add a one transformations
- Search for filter, and select **«Filter fields by name»**
- Remove time Time and wlan_profile_name

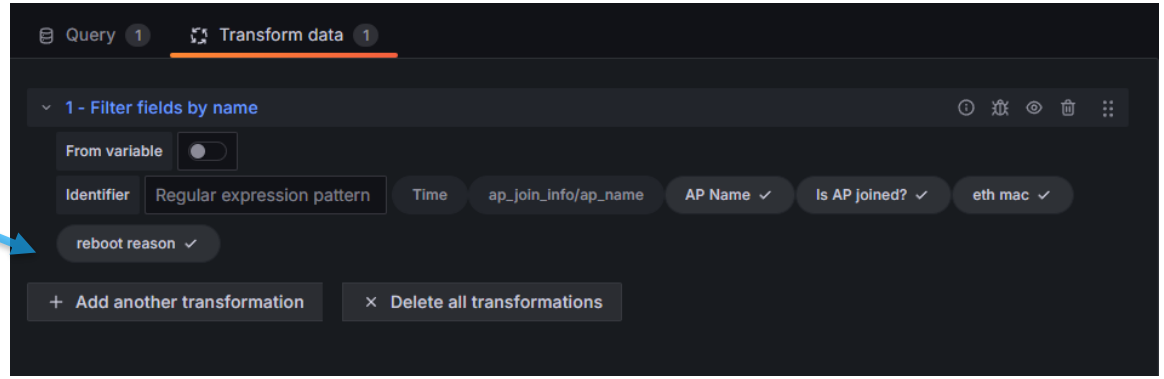


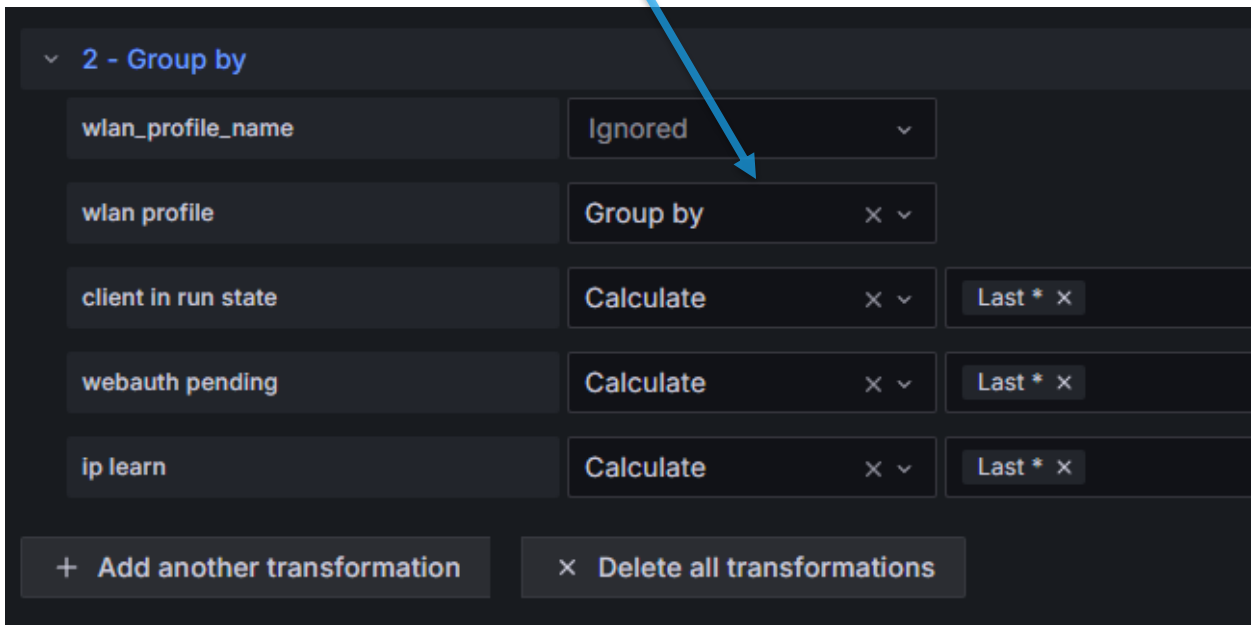


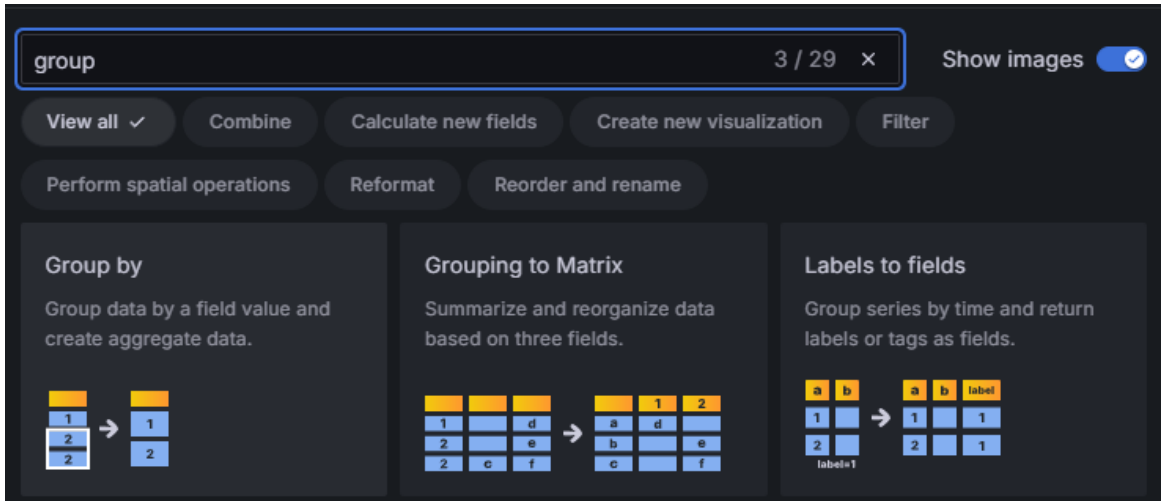


Lab Exercise #31: Grafana – C9800 data (WLAN Stats)

- Add a second transformations
- Search for filter, and select **«Group By»**
  - wlan_profile_name = Ignored
  - wlan profile = Group by
  - All others = Last*





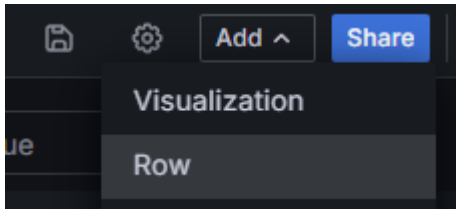


Lab Exercise #31: Grafana – C9800 data (WLAN Stats)

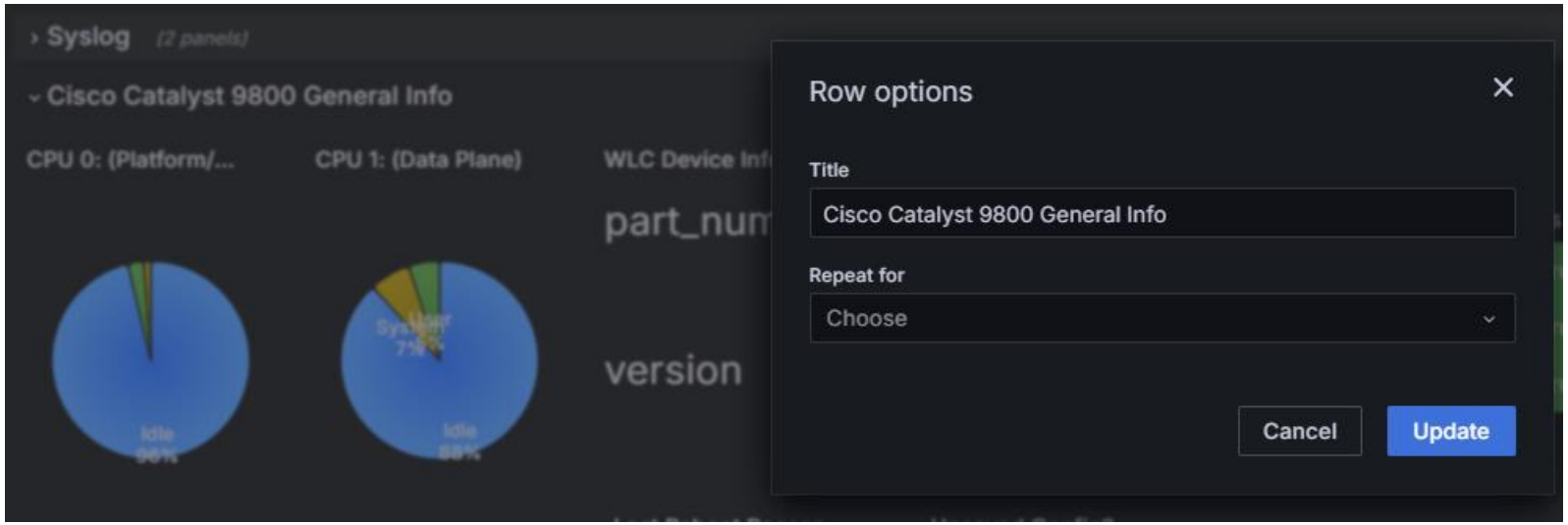
- Add this panel to the right of all your other panels

Lab Exercise #31: Grafana – C9800 data (WLAN Stats)

- Add a new row

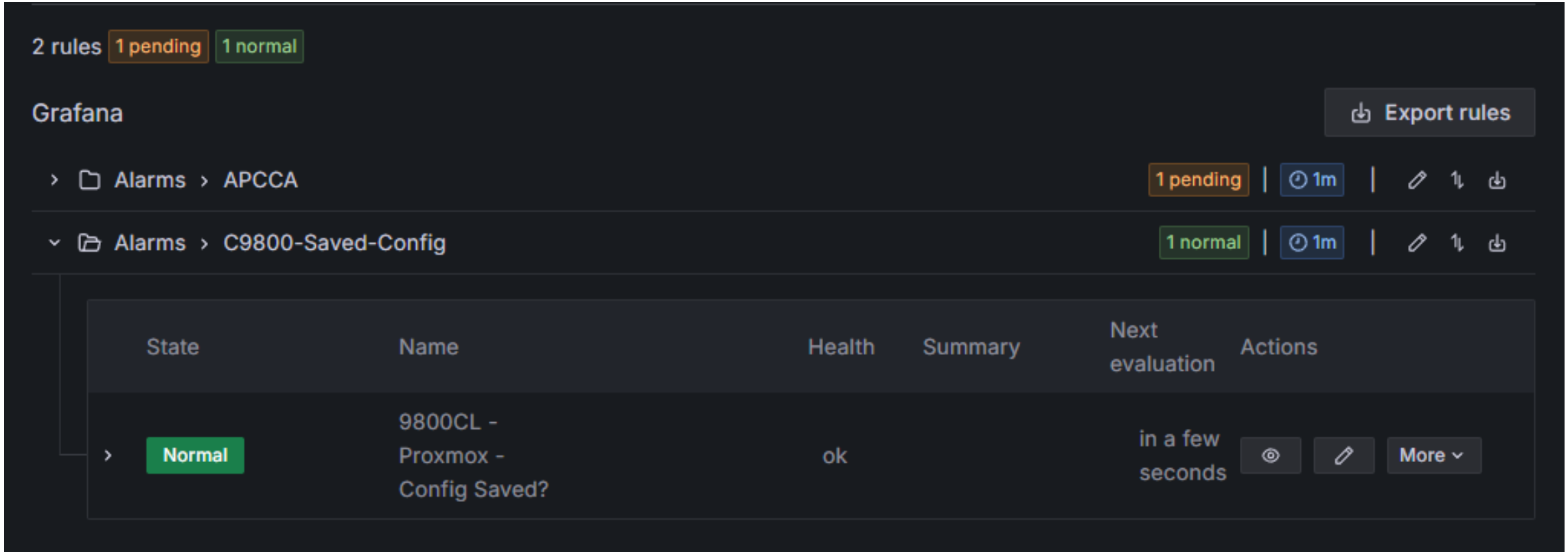


- Give it a name like Cisco Catalyst 9800 General Info
  - Move all your panels below this panel.



Want to make Grafana – Slack Bot alarms?

- Ask Kjetil Teigen Hansen to give you a Demo and explanation ☺



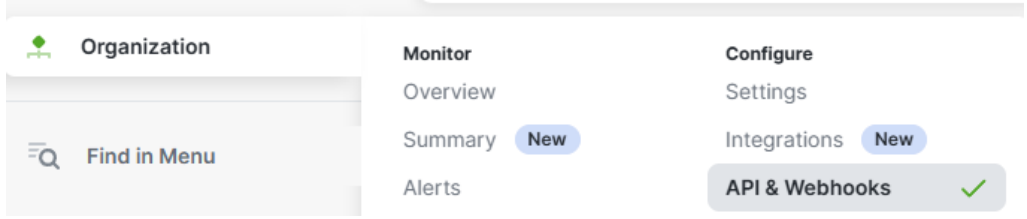
Lab Exercise #60: Using Meraki APIs with Postman

- In this exercise we will explore some introductory variants of Meraki programmability/automation
- Postman: How to use the Meraki Postman API collection
- Python: Create a simple script based on what we explored in Postman
- References:
  - https://documentation.meraki.com/General_Administration/Other_Topics/Cisco_Meraki_Dashboard_API
  - https://developer.cisco.com/meraki/api-v1/getting-started/

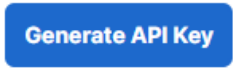
- Note: I have tried to create the Meraki and MIST guides using the same principles. So Exercise 60&70 are mostly the same, and the same for 61&71. We will use the same influxdb database and Grafana dashboard for both solutions, only use some different modules in the Python script

Obtaining API key

- Your API key is generated under Organization -> Configure ->API & Webhooks



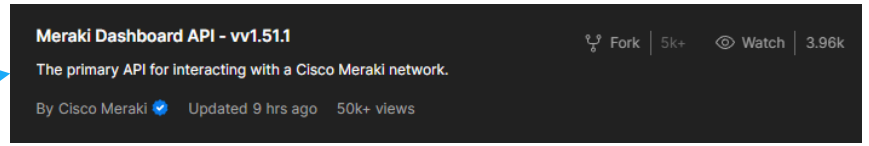
- Then click "Generate API Key"

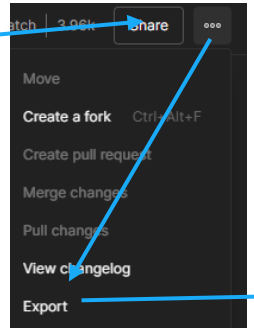


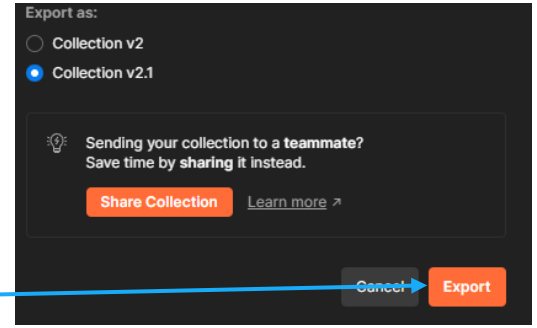
- !!! Notice that the API key is bound to your user (email) and will have the same access as your user. So it is not for the specific Organization or Network, it is the key for your user !!!
- Write down your API key in your password manager. If you lose it you will have to revoke it and create a new
- You can get Read access to the organization that the Meraki router in this lab belong, so you can follow the examples. You can also do the same examples using your own organization of course :)

Postman Meraki API Collection

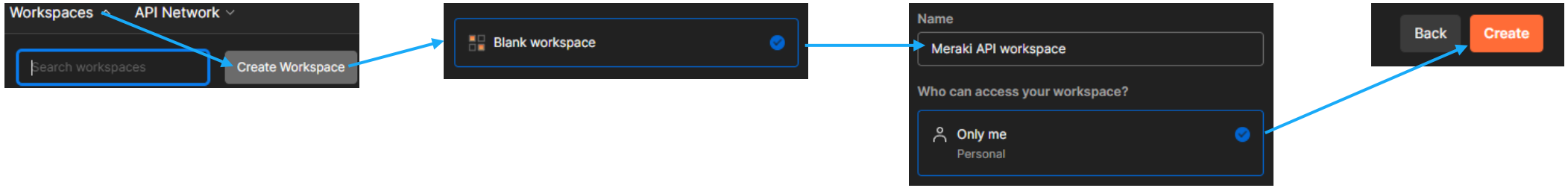
- There is a very nice Collection for Postman to use the Meraki Dashboard API
- https://www.postman.com/meraki-api?tab=collections
- Click the "Meraki Dashboard API - v1.54.0" or whatever version is out when you do this exercise
- Go to the "three dots menu" and choose "Export"
- Choose "Collection v2.1" and "Export"





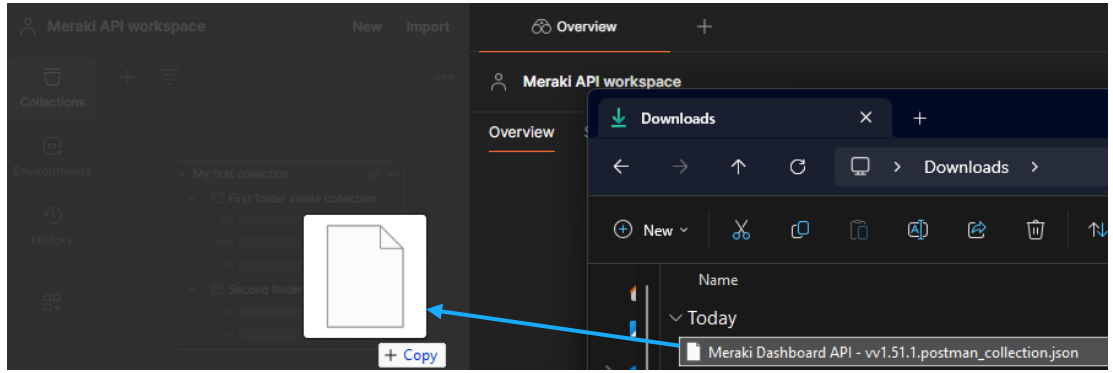


- In Postman, create a new Workspace (or use an existing)

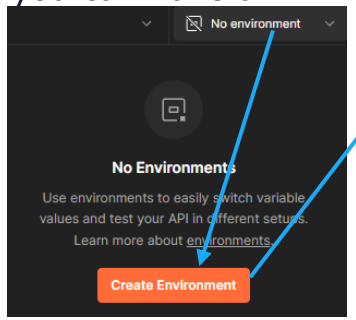


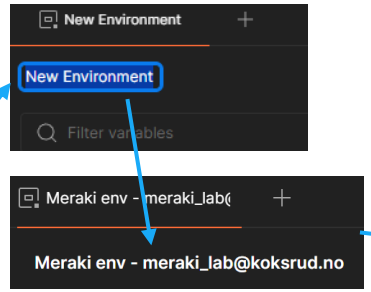
Postman Meraki API Collection

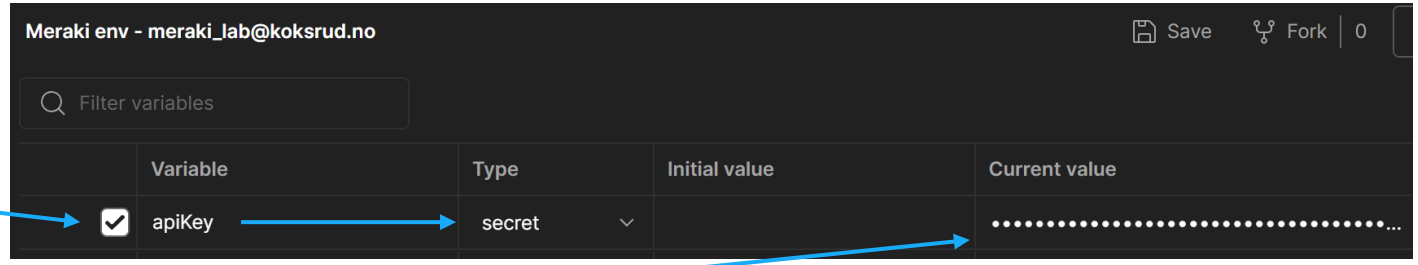
- You import the Meraki API Collection by just drag-drop'ing the JSON file you downloaded, into the "Collections" in Postman



- Start by creating an Environment for your Meraki account. If you have multiple Meraki users, you can have an Environment for each of them. I will create an env for my user "meraki_lab@koksrud.no"



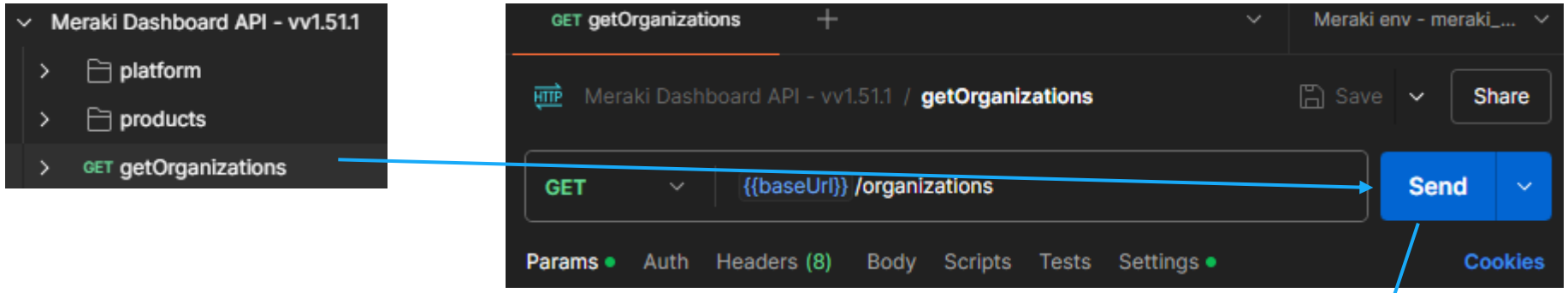




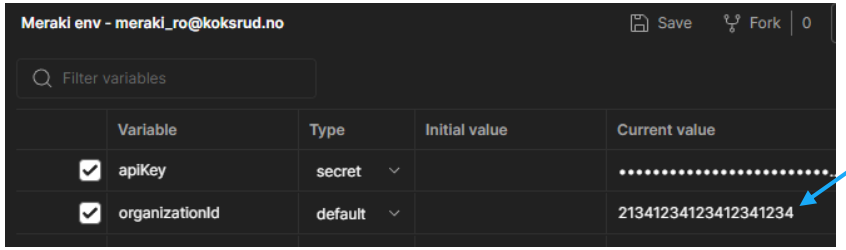
- In that environment, create a variable with name "apiKey" and put YOUR token in "Current Value", then Save.
- Reference: https://www.postman.com/meraki-api/cisco-meraki-s-public-workspace/overview

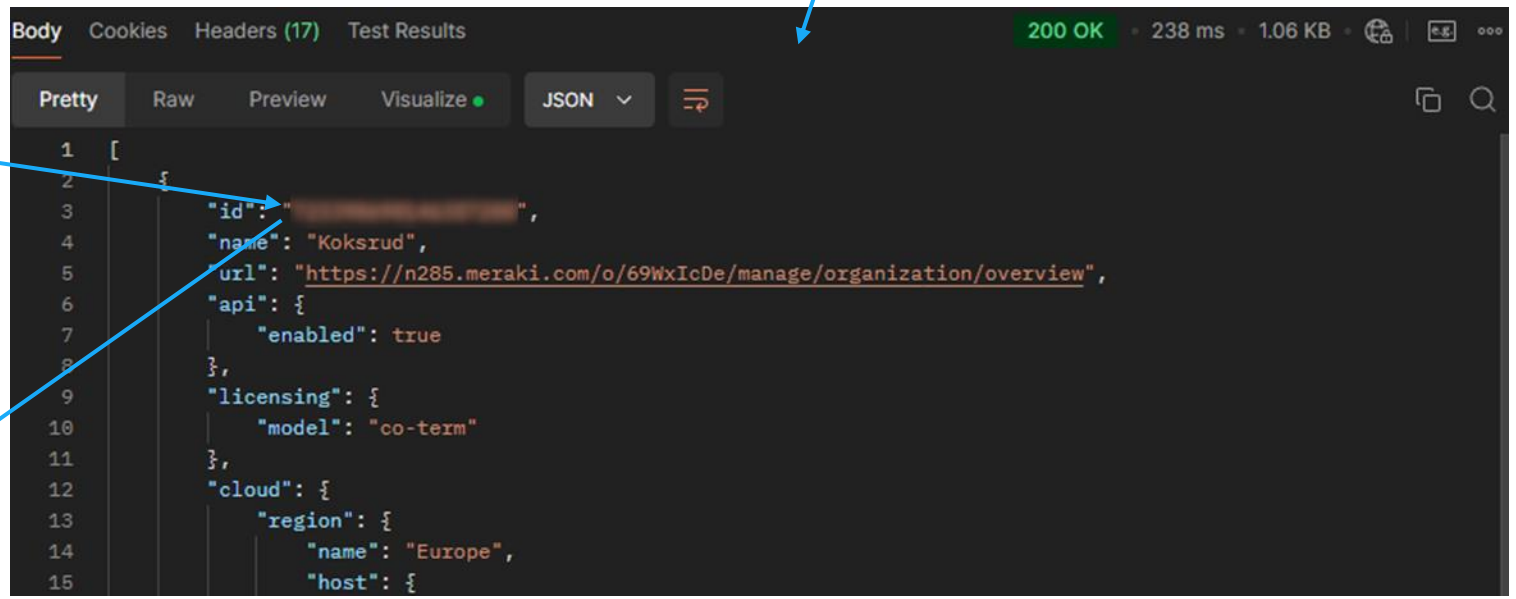
Postman Meraki API Collection

- You can now try out the "getOrganizations" API call



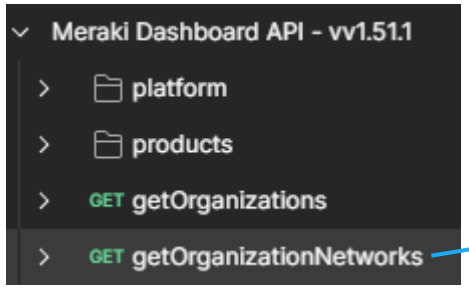
- For further queries to that organization, you can save another environment variable with your organization ID

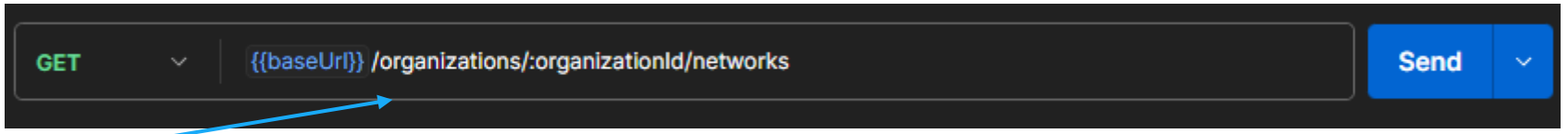


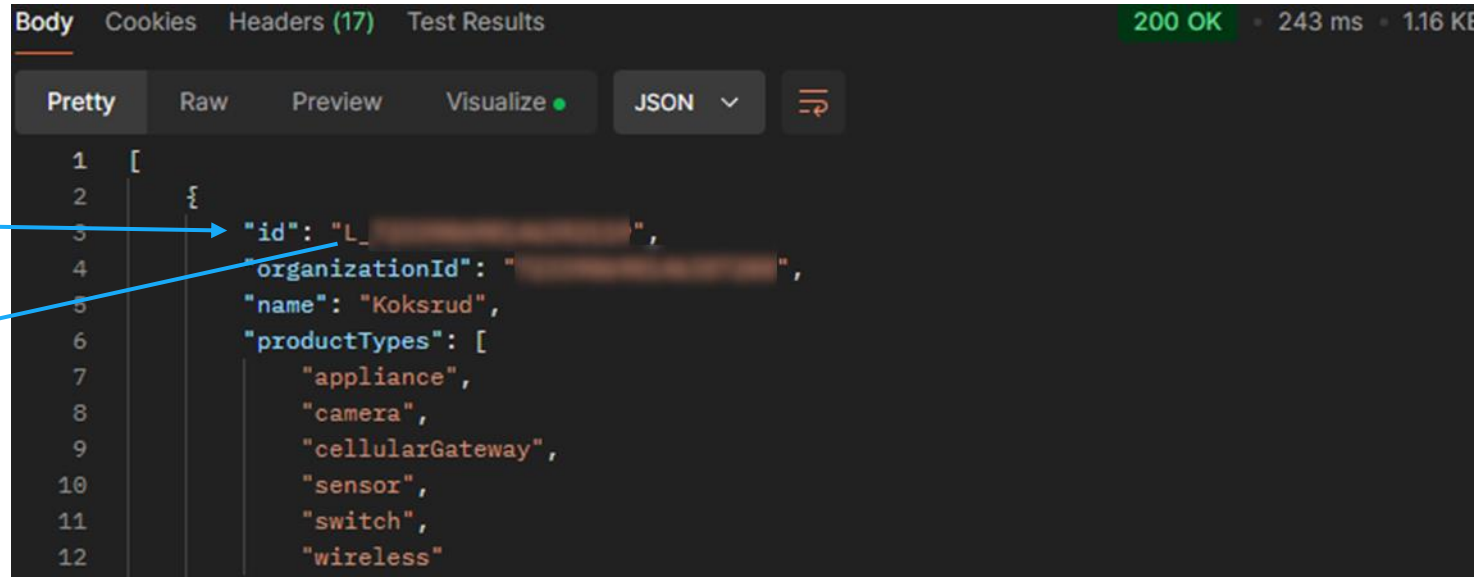


Postman Meraki API Collection

- After getting your Org ID, you can get the Networks

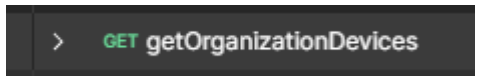






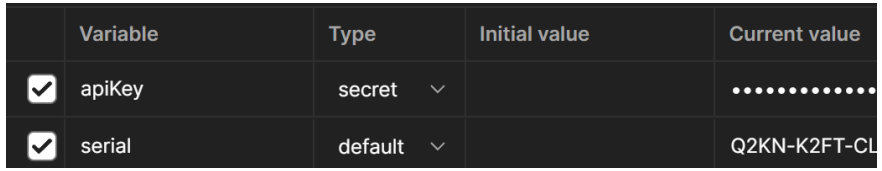
- Save the network ID of your choice (if you have more than one) to an env variable "networkId" as well

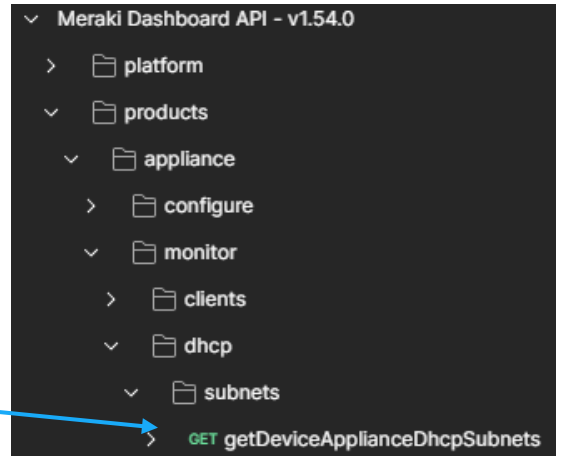
- With this, you can continue to run the other API calls included in the Collection

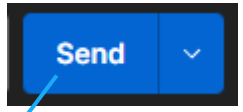


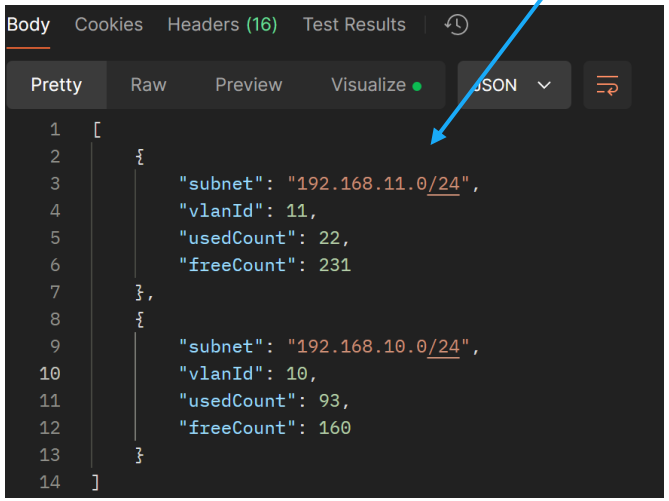
Python - Use the Postman API

- Try using one of the APIs from the Postman collection. In this example we will get the DHCP info from the firewall in this lab. For this request to work you will need to have gathered the serial number of the FW (find it with getOrganizationDevices), and you need your API key. Start by running the request in Postman









- Then click the "code" button </> and select "Python - Requests" to get an example

Python - Use the Meraki API

- Create a new folder "Lab60-Meraki-get-dhcp"
  - Copy the "get-client-table.py" from Lab44 where we have a simple menu structure, and use a .env file. Rename it to "meraki-script.py"
  - Create a .env file, and fill inn your values from the Postman part of the lab
- Get the values from the .env file into variables
- Modify the script to use some modified functions, and show some modified text

```
while True:
    print(f'\nYou are connected to organization {organizationId} and network {networkId}')
    print("Menu:")
    print("1. Show Device summary")
    print("2. Show client summary")
    print("3. Show CDP neighbors")
    print("4. Show DHCP subnets")
    print("5. Quit")
```

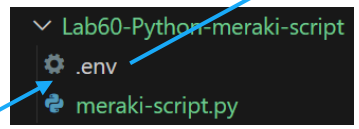
- Here is an example of a "empty" script. All the functions are just printing the headline. Next we will write the RESTCONF calls to the Meraki cloud

Lab60-Python-meraki-script > .env

```
apiKey='e98f2715
organizationId='
networkId='L_656
serial='Q2KN-K2F
```

.env

```
apiKey='{YOUR TOKEN}'
orgId='{YOUR Org ID}'
networkId='{YOUR Network ID}'
serial='{YOUR DEVICE SERIAL}'
```



solutions > Lab60-Python-meraki-script > meraki-script.py > ...

```
from dotenv import dotenv_values

# Import variables from the .env file
env = dotenv_values('.env')
apiKey = env['apiKey']
organizationId=env['orgId']
networkId=env['networkId']
serial=env['serial']
```

meraki-script.py

```
from dotenv import dotenv_values

# Import variables from the .env file
env = dotenv_values('.env')
apiKey = env['apiKey']
organizationId=env['orgId']
networkId=env['networkId']
serial=env['serial']

# Define functions
def show_device_summary():
    print('Device summary:\n-------------------------------------')

def show_client_summary():
    print('Client summary:\n-------------------------------------')

def show_cdp_neighbors():
    print(f"CDP neighbors for device {serial}:\n-------------------------------------")

def get_device_dhcp_subnets():
    print(f"DHCP subnets for device {serial}:\n-------------------------------------")

# Text menu with 4 options that run each of the functions, or quit
while True:
    print(f'\nYou are connected to organization {organizationId} and network {networkId}')
    print("Menu:")
    print("1. Show Device summary")
    print("2. Show client summary")
    print("3. Show CDP neighbors")
    print("4. Show DHCP subnets")
    print("5. Quit")

    choice = input("Enter choice: ")

    if choice == "1":
        show_device_summary()
    elif choice == "2":
        show_client_summary()
    elif choice == "3":
        show_cdp_neighbors()
    elif choice == "4":
        get_device_dhcp_subnets()
    elif choice == "5":
        break
    else:
        print("Invalid choice. Please choose again.")
```


Python - Use the Meraki API

- Merge the code snippet from Postman into the function show_device_dhcp_subnets()
  - Import the requests package (top of file)
  - Write the rest into the get_device_dhcp_subnets() function
  - We change the url with an f-string and put in the variable "serial" from the .env file instead
  - Instead of the Authorization in the Postman example, we use the key "X-Cisco-Meraki-API-Key" and use apiKey from the .env file as the value

```python
import requests
```

```python
def get_device_dhcp_subnets():
    print(f"DHCP subnets for device {serial}:\n----------------------------------
    url = f"https://api.meraki.com/api/v1/devices/{serial}/appliance/dhcp/subnets"
    payload = {}
    headers = {
        'Accept': 'application/json',
        'X-Cisco-Meraki-API-Key': apiKey
    }
    response = requests.request("GET", url, headers=headers, data=payload)
    print(response.text)
```

Postman:

Python - Requests

```python
import requests

url = "https://api.meraki.com/api/v1/devices/[redacted]/appliance/dhcp/subnets"

payload = {}
headers = {
  'Accept': 'application/json',
  'Authorization': '••••••'
}

response = requests.request("GET", url, headers=headers, data=payload)

print(response.text)
```


Python - Use the Meraki API

- Do the same for show_device_summary()
  Use getOrganizationDevices from Postman
  Change the OrgId from Postman with {organizationId} from .env

```
def show_device_summary():
    print('Device summary:\n-------------------------------------')
    url = f"https://api.meraki.com/api/v1/organizations/{organizationId}/devices"
    payload = {}
    headers = {
        'Accept': 'application/json',
        'X-Cisco-Meraki-API-Key': apiKey
    }
    response = requests.request("GET", url, headers=headers, data=payload)
    print(response.text)
```

- Do the same for show_client_summary()
  Use getNetworkClients from Postman
  Change the Network ID from Postman with {networkId} from .env

```
def show_client_summary():
    print('Client summary:\n-------------------------------------')
    url = f"https://api.meraki.com/api/v1/networks/{networkId}/clients"
    payload = {}
    headers = {
        'Accept': 'application/json',
        'X-Cisco-Meraki-API-Key': apiKey
    }
    response = requests.request("GET", url, headers=headers, data=payload)
    print(response.text)
```

- Do the same for show_cdp_neighbors()
  Use getDeviceLldpCdp from Postman
  Change the S/N from Postman with {serial} from .env

```
def show_cdp_neighbors():
    print(f"CDP neighbors for device {serial}:\n-------------------------------------")
    url = f"https://api.meraki.com/api/v1/devices/{serial}/lldpCdp"
    payload = {}
    headers = {
        'Accept': 'application/json',
        'X-Cisco-Meraki-API-Key': apiKey
    }
    response = requests.request("GET", url, headers=headers, data=payload)
    print(response.text)
```


Python - Use the Meraki API

- Some suggestions for next steps could be
  - Print the JSON output more pretty, you can use pretty-print (from pprint import pprint)
  - Save the JSON to a JSON file (use the method from Lab 40/44 where you saved run-config)
  - Flatten the JSON output to a pandas dataframe and save to CSV or Excel (From Lab 41)

Lab Exercise #61: Python - Meraki->InfluxDB->Grafana

- This lab exercise will be the preparation for a Grafana dashboard visualizing data from our Meraki APs.We will write a simple Python script that
  1. Pulls data from Meraki APs
  2. Inserts the data in InfluxDB
- The Grafana dashboard itself will be vendor independent, the same dashboard will be used in the MIST exercise (Lab Exercise #71)
- In this exercise we will build on the previous exercise by using the RESTCONF calls that we can figure out by using the Meraki API collection in Postman. You can start by creating requests in Postman for the following
  - getOrganizationDevices (platform>configure>devices)
  - getOrganizationWirelessDevicesChannelUtilizationByDevice (products>wireless>monitor>devices>channelUtilization>byDevice)

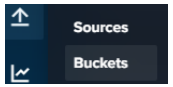
Prerequisites - Python packages

- To prepare we will install some Python packages in our "automation-venv" virtual environment
  - InfluxDB package
    - References: https://github.com/influxdata/influxdb-python
    - References: https://influxdb-python.readthedocs.io/en/latest/

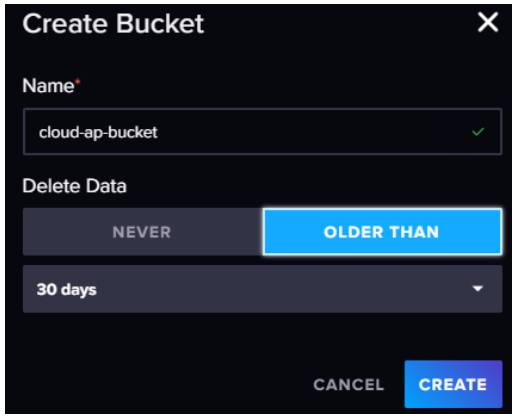
```
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab61_Python_Meraki_InfluxDB$ pip install influxdb
```


Prerequisites - InfluxDB databases

- We will also create a new bucket + database in InfluxDB
- Login to InfluxDB - http://{ubuntu-devnet}:8086/
- Go to Buckets



- Create a new bucket "cloud-ap-bucket"



- Create the database for this bucket (so we can use it with InfluxQL)

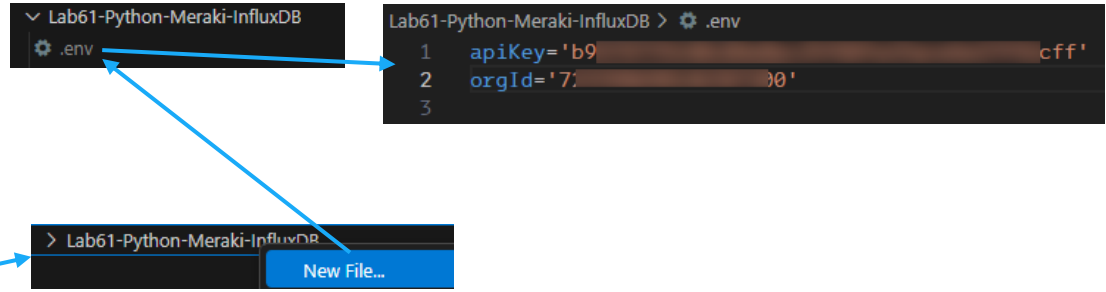
```
devnet-adm@ubuntu-devnet:~/tig-stack$ docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED       STATUS       PORTS
56c9135dbb27   influxdb:2.7.10        "/entrypoint.sh infl…"   5 hours ago   Up 5 hours   0.0.0.0:8086->8086/tcp, :::8086->8086/tcp
devnet-adm@ubuntu-devnet:~/tig-stack$ docker exec -it <your_container-id> bash
root@56c9135dbb27:/# influx v1 dbrp list --org-id <your_org-id> --token <your_admin_token>
ID      Database        Bucket ID       Retention Policy        Default Organization ID
(…)
cf94ebbf115524c3        cloud-ap-bucket cf94ebbf115524c3        autogen                 true    f8f6ee5d500865de

root@56c9135dbb27:/# influx v1 dbrp create --db cloud-ap-db --rp cloud-ap-rp --bucket-id cf94ebbf115524c3 --default --org-id <your_org-id> --token <your_admin_token>

ID                      Database        Bucket ID               Retention Policy        Default Organization ID
0df9b97891720000        cloud-ap-db     cf94ebbf115524c3        cloud-ap-rp             true    f8f6ee5d500865de

root@56c9135dbb27:/# exit
devnet-adm@ubuntu-devnet:~/tig-stack$
```


Pulling Meraki data using Python

- Now we will create the Python script that pulls data from the Meraki API, and inserts this into our influxdb database
- Start by creating the .env file. You can copy this from the previous exercise, but you will only need the apiKey and orgId for this exercise
- Insert your apiKey and orgId into the file
- !! This script assumes that you have one or more Meraki APs in your organization !!

.env

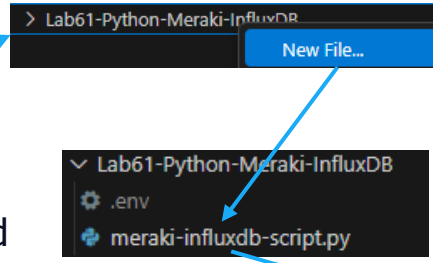
```
apiKey='{YOUR TOKEN}'
orgId='{YOUR Org ID}'
```



Pulling Meraki data using Python

- Create the Python file "meraki-influxdb-script.py"
- In this exercise you can copy-paste the example to the right, and we will explain it part by part in the next slides
- Output should be something like this

```
devnet-adm in ubuntu-devnet in wifi-automation/Lab61-Python-Meraki-Influ
❯ python3 meraki-influxdb-script.py
Updated InfluxDB with data from APU01-Kjellerstue at 2025-01-27T17:02:08Z
Updated InfluxDB with data from AP201-2.etg at 2025-01-27T17:02:08Z
Updated InfluxDB with data from AP102-Gang at 2025-01-27T17:02:08Z
Updated InfluxDB with data from AP101-Kjøkken at 2025-01-27T17:02:08Z
```



.env

```
apiKey='{YOUR TOKEN}'
orgId='{YOUR Org ID}'
```

meraki-influxdb-script.py

```
from influxdb import InfluxDBClient
from datetime import datetime, timezone
from dotenv import dotenv_values
import requests
import json
import time

# Import variables from the .env file
env = dotenv_values('.env')
apiKey = env['apiKey']
organizationId=env['orgId']

def get_devices():
    url = f"https://api.meraki.com/api/v1/organizations/{organizationId}/devices"
    payload = {}
    headers = {
        'Accept': 'application/json',
        'X-Cisco-Meraki-API-Key': apiKey
    }
    response = requests.request("GET", url, headers=headers, data=payload)
    return response.text

def get_channel_utilization():
    url =
f"https://api.meraki.com/api/v1/organizations/{organizationId}/wireless/devices/channelUtilization/byDevice?
timespan=300&interval=300"
    payload = {}
    headers = {
        'Accept': 'application/json',
        'X-Cisco-Meraki-API-Key': apiKey
    }
    response = requests.request("GET", url, headers=headers, data=payload)
    return response.text

def update_influxDB(measurement_name, hostname, fields):
    timestamp = datetime.now(timezone.utc).strftime('%Y-%m-%dT%H:%M:%SZ')
    json_body = [
        {
            "measurement": measurement_name,
            "tags": {
                "host": hostname,
            },
            "time": timestamp,
            "fields": fields
        }
    ]
    client = InfluxDBClient('localhost', 8086, 'devnet-adm', 'ChangeMe2025!', 'cloud-ap-db')
    client.write_points(json_body)
    print(f"Updated InfluxDB with data from {hostname} at {timestamp}")

# Iterate through all devices and do stuff
def iterate_access_points():
    devices = json.loads(get_devices())
    utilization = json.loads(get_channel_utilization())
    ap_serial_map = {}
    interference = {}
    for device in devices:
        if device['productType'] == "wireless":
            ap_serial_map[device['serial']] = device['name']
    for util in utilization:
        serial = util['serial']
        interference[serial] = {}
        for band in util['byBand']:
            interference[serial][band['band']+"GHz Wi-Fi"] = band['wifi']['percentage']
            interference[serial][band['band']+"GHz non-Wi-Fi"] = band['nonWifi']['percentage']
            interference[serial][band['band']+"GHz Total"] = band['total']['percentage']
        update_influxDB('ch_utilization', ap_serial_map[serial], interference[serial])

while True:
    iterate_access_points()
    time.sleep(10)
```


Pulling Meraki data using Python

- Imports at the start of the file
  - InfluxDB module, to communicate with influxdb
  - Datetime, to timestamp our values
  - dotenv to import values from the .env file
  - requests to use REST APIs
  - JSON module, to parse the output from API calls, to use them for input to influxdb
  - time, we use this to pause the script for 10 seconds before we poll the data once more
- Import the .env file variables

Lab61-Python-Meraki-InfluxDB > meraki-influxdb-script.py > ...

```
1  from influxdb import InfluxDBClient
2  from datetime import datetime, timezone
3  from dotenv import dotenv_values
4  import requests
5  import json
6  import time
7
8  # Import variables from the .env file
9  env = dotenv_values('.env')
10 apiKey = env['apiKey']
11 organizationId=env['orgId']
```


Pulling Meraki data using Python

- get_devices() function
  - REST API call to Meraki, to get all your devices
  - The response text is returned, and saved to where you call the function from
  - Will contain all the orgs devices (APs, FWs, switches etc)
- get_channel_utilization() function
  - REST API call to Meraki, to get the channel utilization for each AP in your organization. It will get both Wi-Fi, non-Wi-Fi and Total Channel Utilization, for 2.4, 5 and 6GHz
  - Note that the long line is cut, see the full line in VS Code
- update_influxDB() function
  - Function to write values to your local influxDB
  - We create a timestamp to go with the measurement
  - We will get measurement_name, hostname and fields from the get_channel_utilization function and use them when we call this function
  - Create the InfluxDBClient object, on the local machine, we will use this to write to influxdb

```python
def get_devices():
    url = f"https://api.meraki.com/api/v1/organizations/{organizationId}/devices"
    payload = {}
    headers = {
        'Accept': 'application/json',
        'X-Cisco-Meraki-API-Key': apiKey
    }
    response = requests.request("GET", url, headers=headers, data=payload)
    return response.text

def get_channel_utilization():
    url = f"https://api.meraki.com/api/v1/organizations/{organizationId}/wireless/devices/channelUtil
    payload = {}
    headers = {
        'Accept': 'application/json',
        'X-Cisco-Meraki-API-Key': apiKey
    }
    response = requests.request("GET", url, headers=headers, data=payload)
    return response.text

def update_influxDB(measurement_name, hostname, fields):
    timestamp = datetime.now(timezone.utc).strftime('%Y-%m-%dT%H:%M:%SZ')
    json_body = [
        {
            "measurement": measurement_name,
            "tags": {
                "host": hostname,
            },
            "time": timestamp,
            "fields": fields
        }
    ]
    client = InfluxDBClient('localhost', 8086, 'devnet-adm', 'ChangeMe2025!', 'cloud-ap-db')
    client.write_points(json_body)
    print(f"Updated InfluxDB with data from {hostname} at {timestamp}")
```

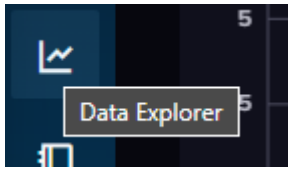

Pulling Meraki data using Python

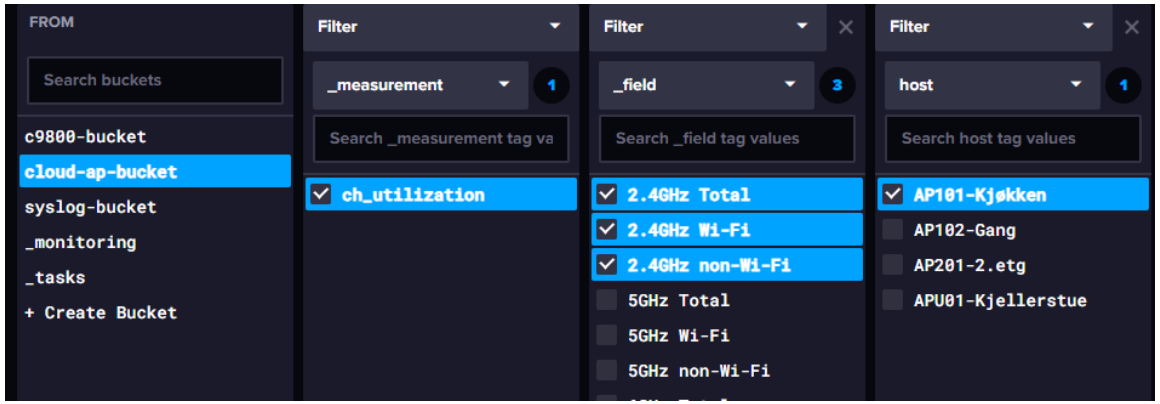
- iterate_access_points() function
  - Get your devices using the get_devices() function. Convert to JSON object and ave to variable "devices"
  - Get channel utilization for all APs using the get_channel_utilization() function. Convert to JSON object and save to variable "utilization"
  - Iterate over all devices in the "devices" list, and save the ones where "productType" = "wireless" to a dictionary mapping the serial numbers to the AP Names
  - Iterate over all values in the "utilization" list. Populate a dictionary "interference" where you use the serial number as a key
  - Update InfluxDB for each serial number
- Use a while loop, that loops forever (until you Ctrl+Z), and sleep for 10 seconds each time you have run the iterate_access_points() function

```
49  # Iterate through all devices and do stuff
50  def iterate_access_points():
51      devices = json.loads(get_devices())
52      utilization = json.loads(get_channel_utilization())
53      ap_serial_map = {}
54      interference = {}
55      for device in devices:
56          if device['productType'] == "wireless":
57              ap_serial_map[device['serial']] = device['name']
58      for util in utilization:
59          serial = util['serial']
60          interference[serial] = {}
61          for band in util['byBand']:
62              interference[serial][band['band']+"GHz Wi-Fi"] = band['wifi']['percentage']
63              interference[serial][band['band']+"GHz non-Wi-Fi"] = band['nonWifi']['percentage']
64              interference[serial][band['band']+"GHz Total"] = band['total']['percentage']
65          update_influxDB('ch_utilization', ap_serial_map[serial], interference[serial])
66
67  while True:
68      iterate_access_points()
69      time.sleep(10)
```

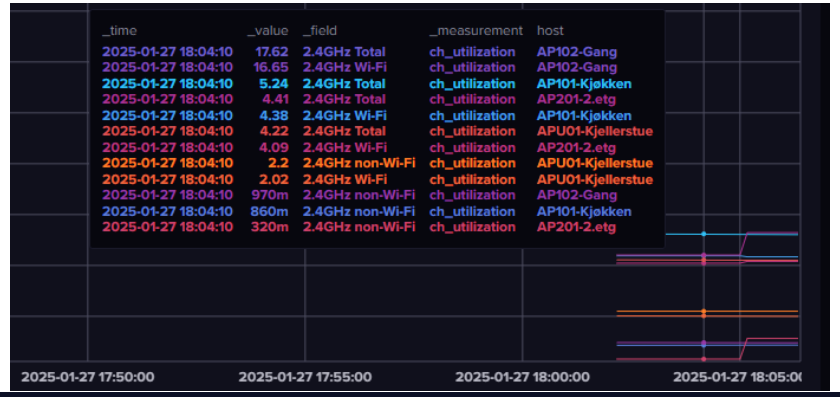

Checking the data in InfluxDB

- Log in to influxdb at http://{UBUNTU_IP}:8086
- Go to Data Explorer, choose cloud-ap-bucket and verify if there is some populated data there





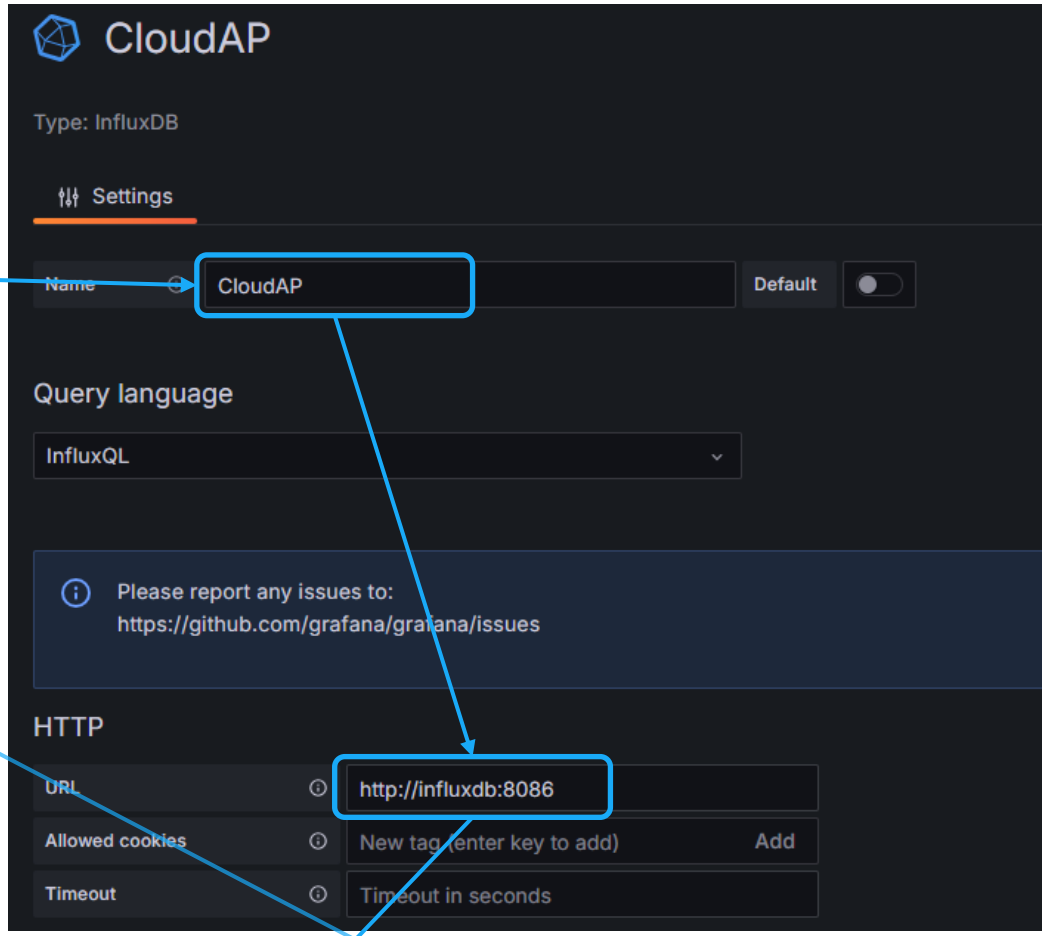
- Submit and see if the graph shows some values

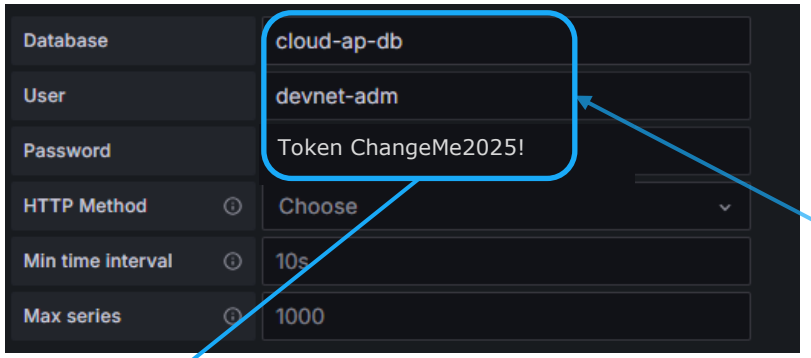


Fikse slides
Export JSON av example dashboard

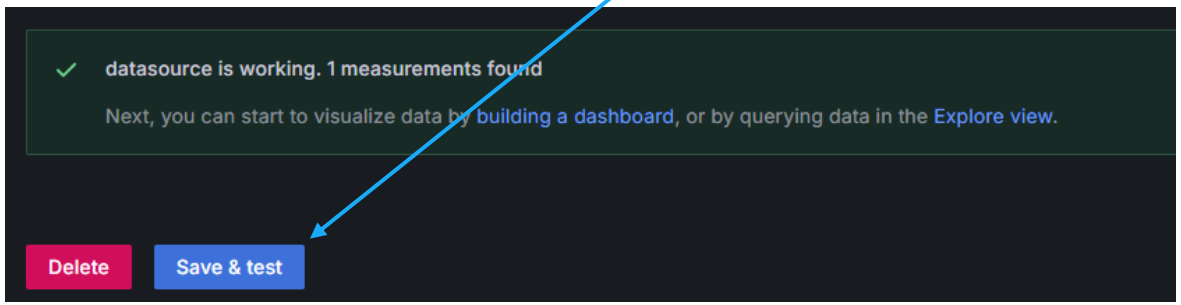
Creating Datasource in Grafana

- Login to Grafana - http://{ubuntu-devnet}:3000/
- Create a new datasource > InfluxDB
- The reason we create a new datasource, is to select a different database than the one we used for the 9800 telemetry



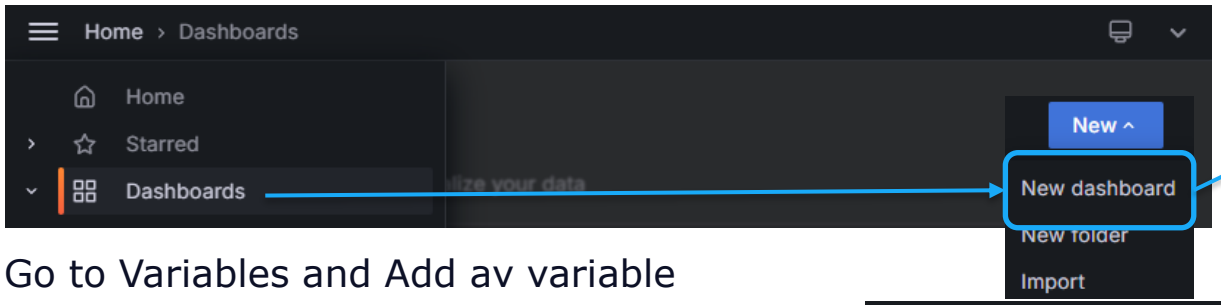


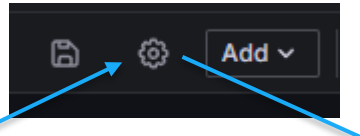
- "Save & test" should give 1 measurement

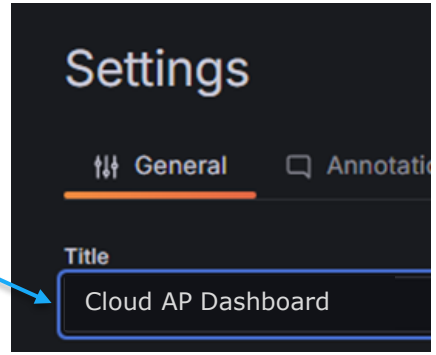


Creating the dashboard in Grafana

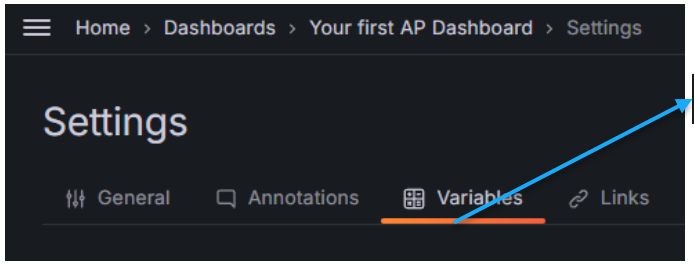
- Create a new Dashboard. Name it "Cloud AP Dashboard" or something

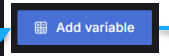


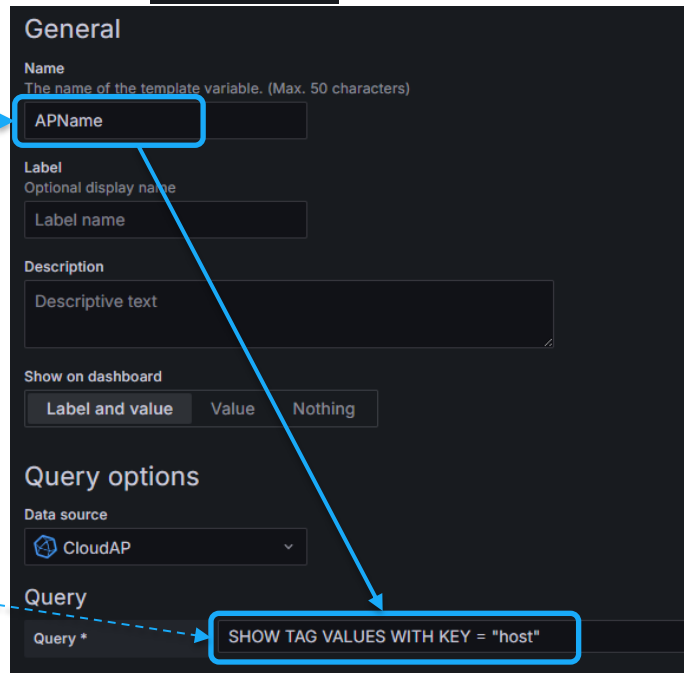




- Go to Variables and Add av variable



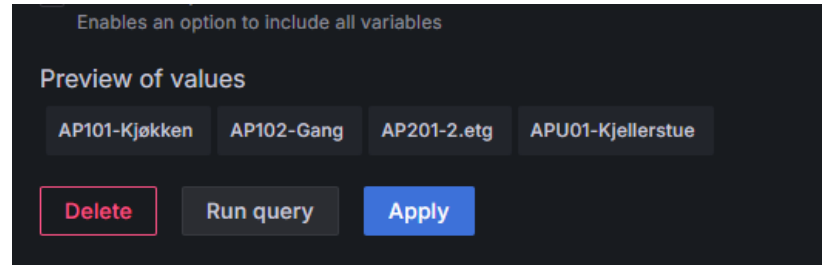




- We show values with key "host". The reason for this is that we put data in influxDB with that key

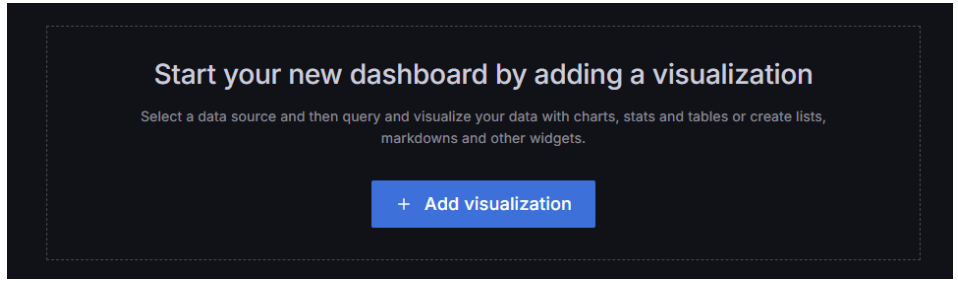
```
json_body = [
    {
        "measurement": measurement_name,
        "tags": {
            "host": hostname,
        },
        "time": timestamp,
        "fields": fields
    }
]
```

- When you press "Run query" you should see the names of your APs that you have inserted into influxdb

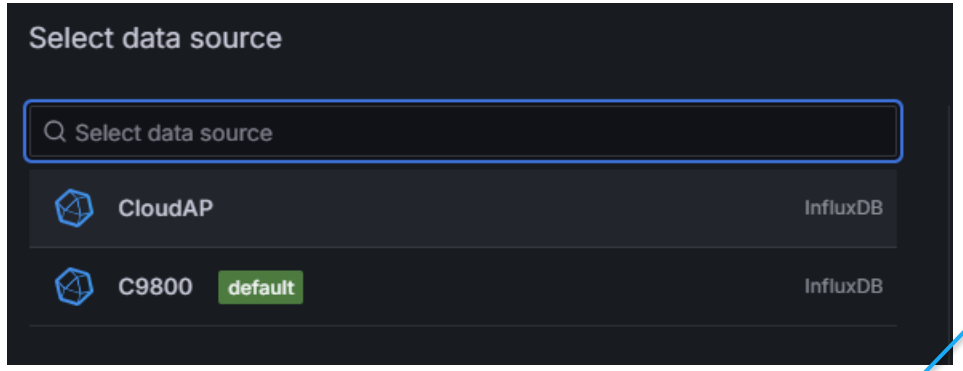


Creating the dashboard in Grafana

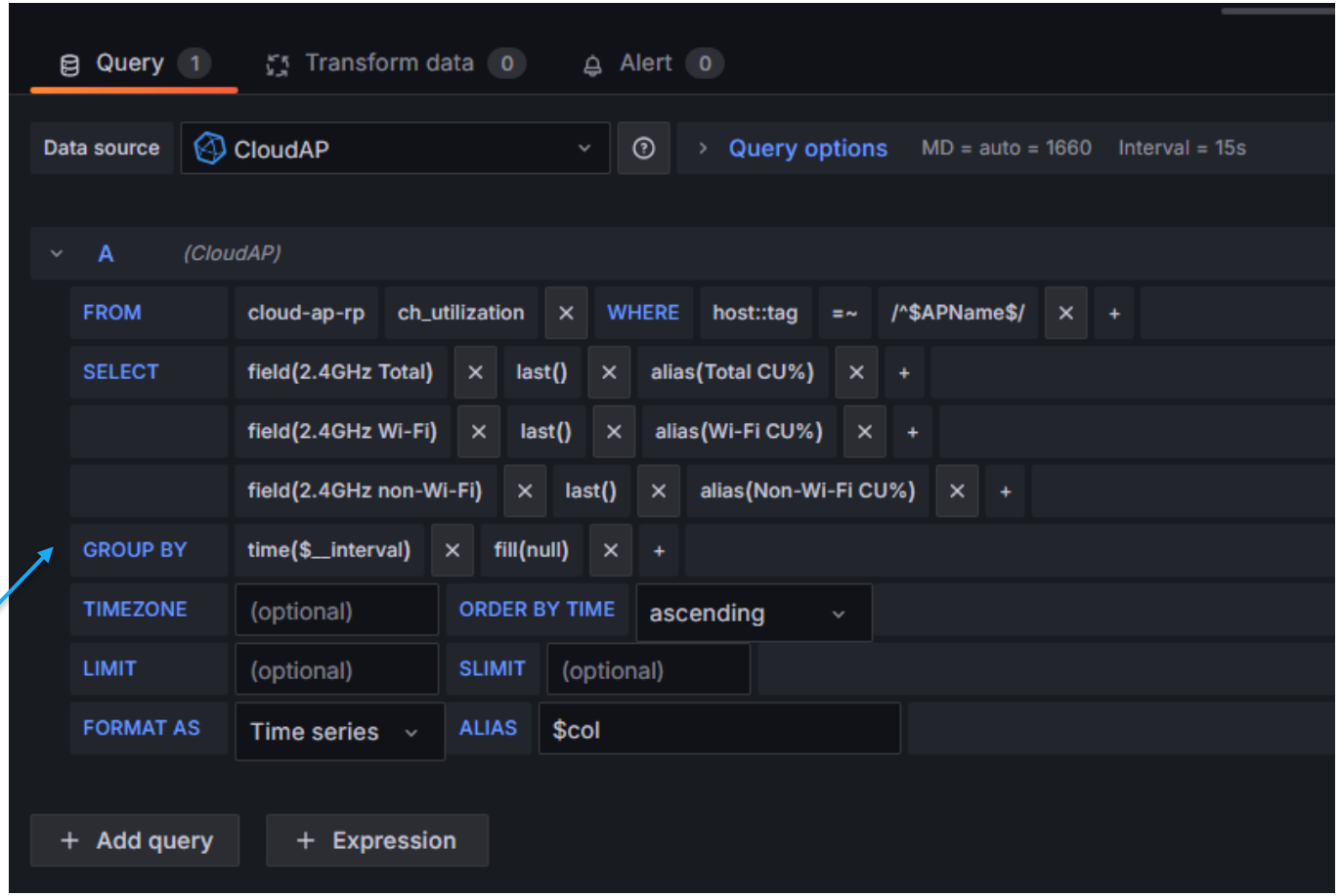
- Go to the dashboard and select "Add visualization"



- Select the "CloudAP" datasource that you created

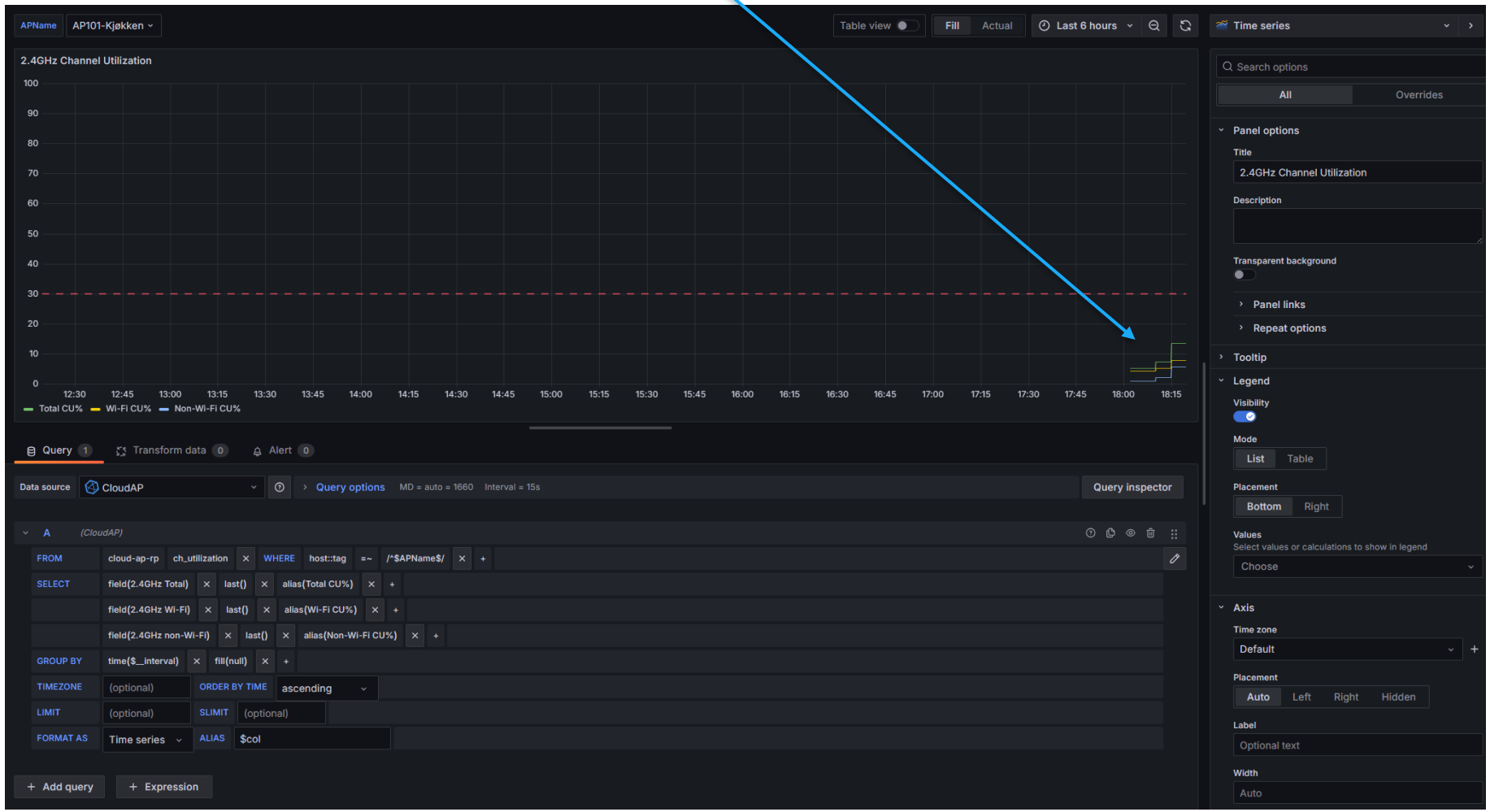


- Replicate this query for your first visualization



Creating the dashboard in Grafana

- You should see the graphs starting to show

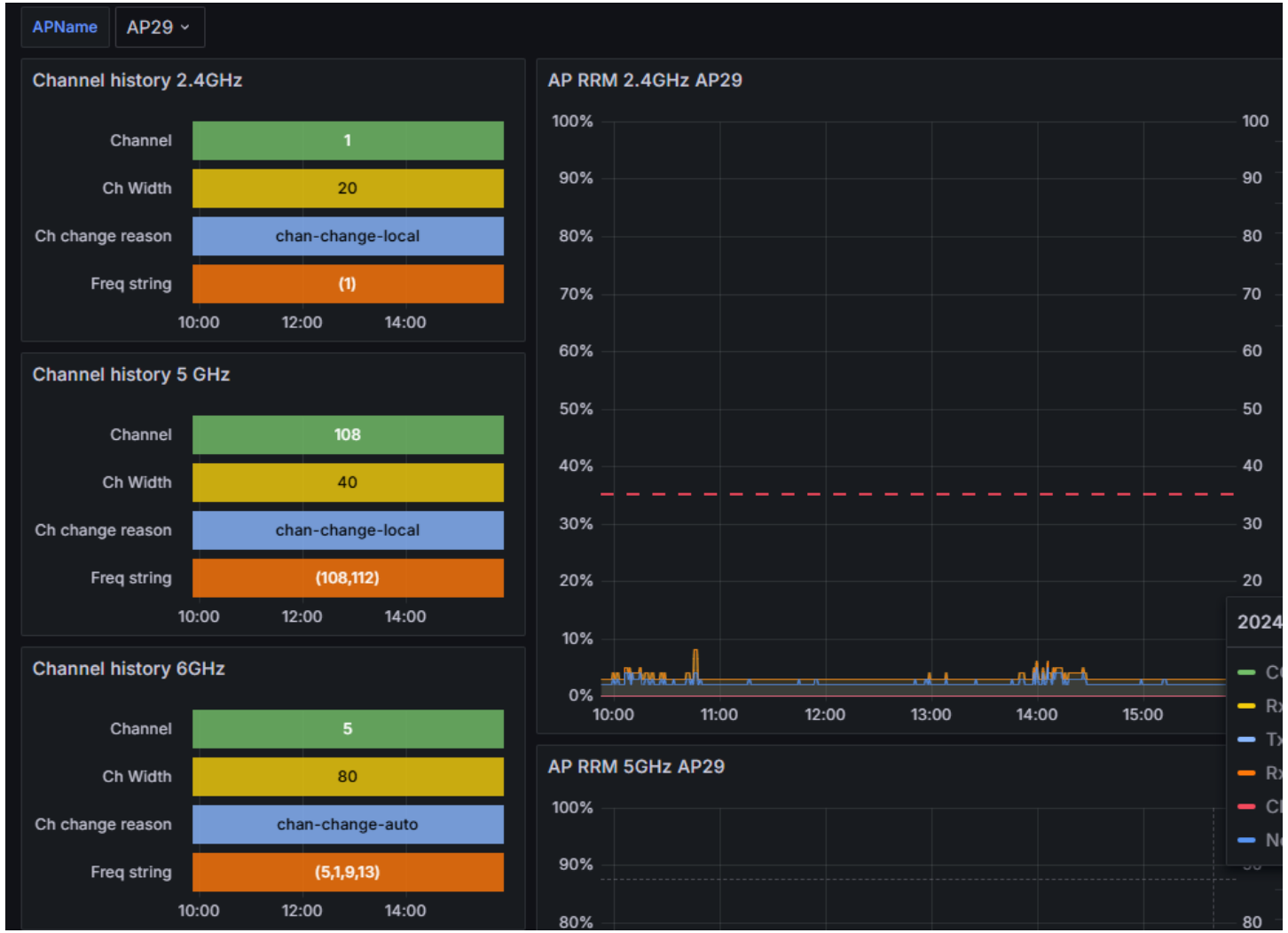


Example dashboard in Grafana

- Duplicate the 2.4GHz visualization and create a panel for 5GHz and 6GHz

Creating the dashboard in Grafana

- Next step suggestions
  - Client count
  - Current channel
  - Channel width
  - etc, check the 9800 dashboard for example



Lab Exercise #70: Using MIST APIs with Postman

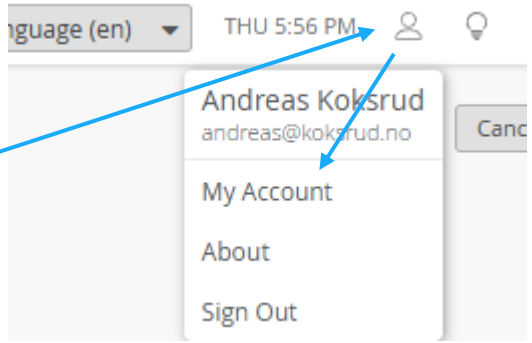
- In this exercise we will explore some introductory variants of MIST programmability/automation
- Postman: How to use the MIST Postman API collection
- Python: Create a simple script based on what we explored in Postman

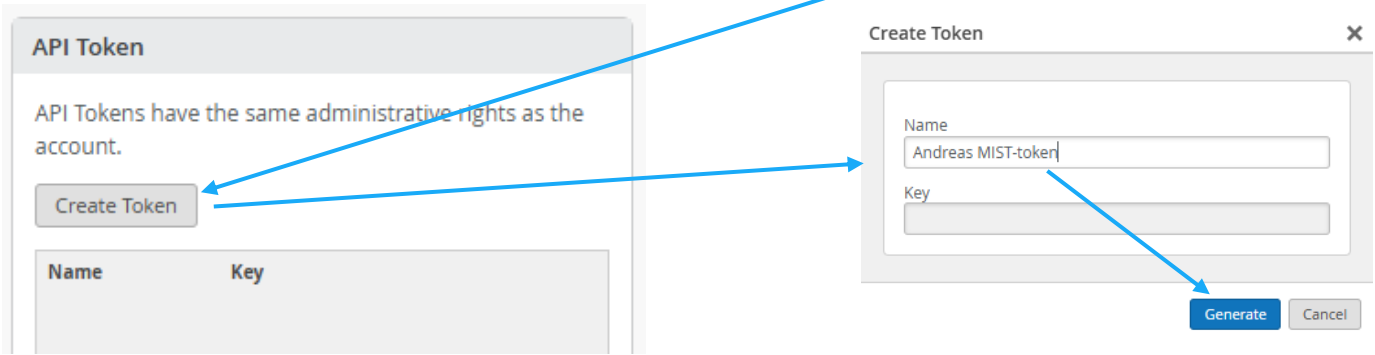
- References:
  - Juniper-MIST API documentation: https://www.juniper.net/documentation/product/us/en/mist/#cat=for_developers
  - https://www.mist.com/documentation/using-postman/
  - MIST-Lab demo apps: https://apps.mist-lab.fr/
  - MIST API Sandbox lab: https://api-class.mist.com/

- Note: I have tried to create the Meraki and MIST guides using the same principles. So Exercise 60&70 are mostly the same, and the same for 61&71. We will use the same influxdb database and Grafana dashboard for both solutions, only use some different modules in the Python script

Obtaining API key

- Start by logging in to your MIST account: https://manage.mist.com
- Click on the user menu in the top right corner and select "My Account"
- In the "API Token" section, select "Create Token"

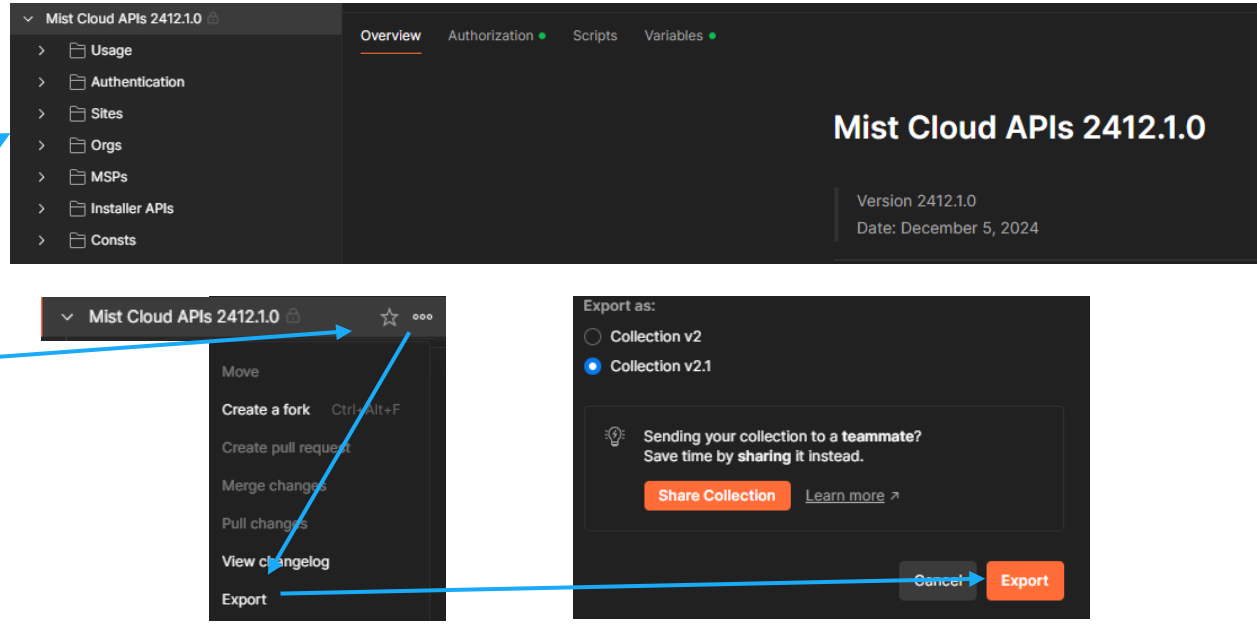




- !!! Notice that the API key is bound to your user (email) and will have the same access as your user. So it is not for the specific Organization or Network, it is the key for your user !!!
- Write down your API key in your password manager. If you lose it you will have to revoke it and create a new
- It is also possible to create Org API keys (https://www.juniper.net/documentation/us/en/software/mist/automation-integration/topics/task/create-token-for-rest-api.html)

Postman MIST API Collection

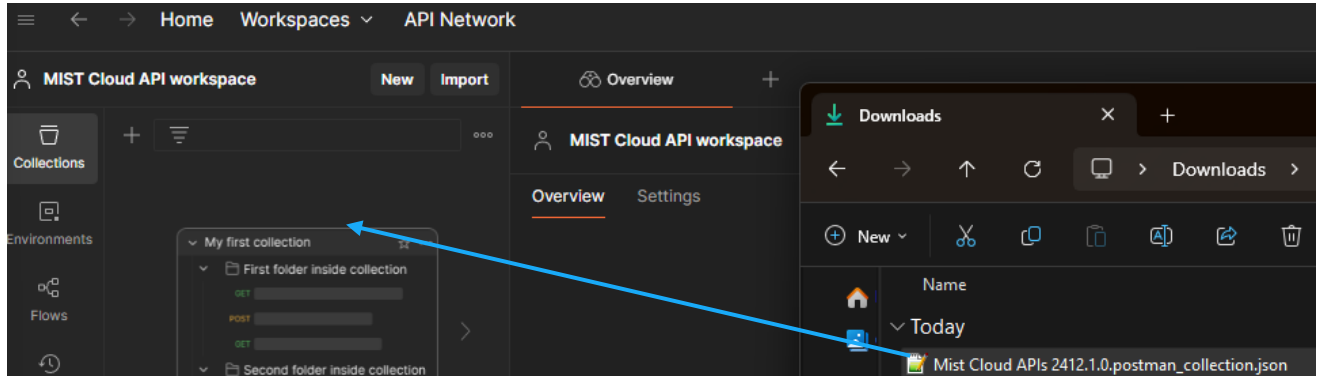
- There is a very nice Collection for Postman to use the MIST Cloud APIs
- https://www.postman.com/juniper-mist/mist-systems-s-public-workspace/overview
- Click the "MIST Cloud APIs 2412.1.0" or whatever version is out when you do this exercise
- Go to the "three dots menu" and choose "Export"
- Choose "Collection v2.1" and "Export"



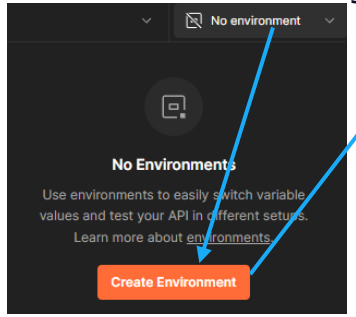
- In Postman, create a new Workspace (or use an existing)

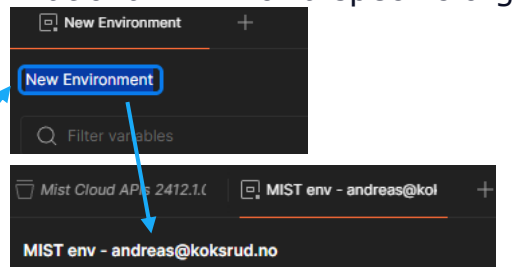
Postman MIST API Collection

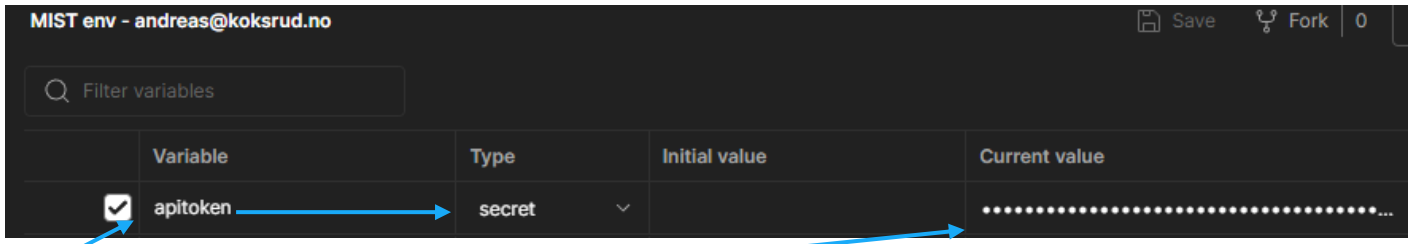
- You import the MIST Cloud API Collection by just drag-drop'ing the JSON file you downloaded, into the "Collections" in Postman



- Start by creating an Environment for your MIST account. If you have multiple MIST users, you can have an Environment for each of them. You can also have an "Organizational API" for a specific org. I will create an env for my user "andreas@koksrud.no"



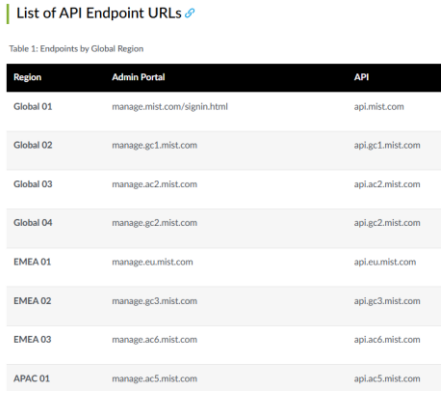


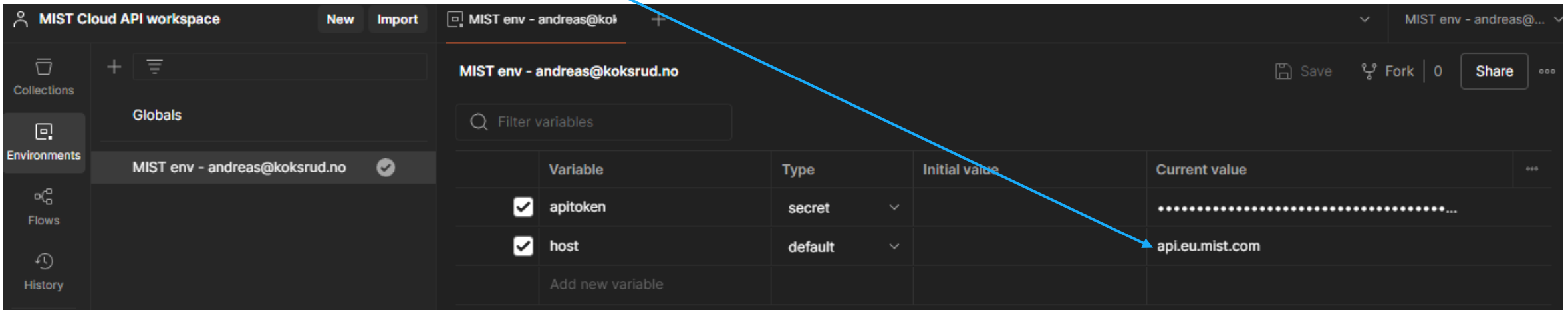


- In that environment, create a variable with name "apitoken" and put YOUR token in "Current Value", then Save. The variable name "apitoken" is used in the MIST Postman Collection so it is easiest to just use that.

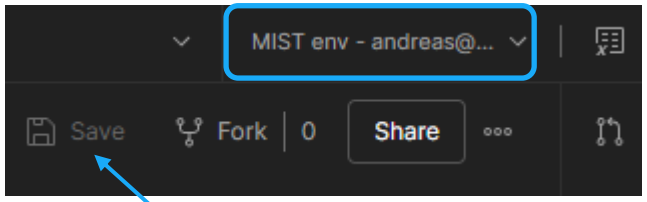
Postman MIST API Collection

- Also create a variable "host"
  - If you log in to MIST web managemet using "manage.mist.com", then put "api.mist.com" as your host
  - If you use "manage.eu.mist.com", then put "api.eu.mist.com", etc... it is documented in [Create API Tokens | Mist | Juniper Networks]





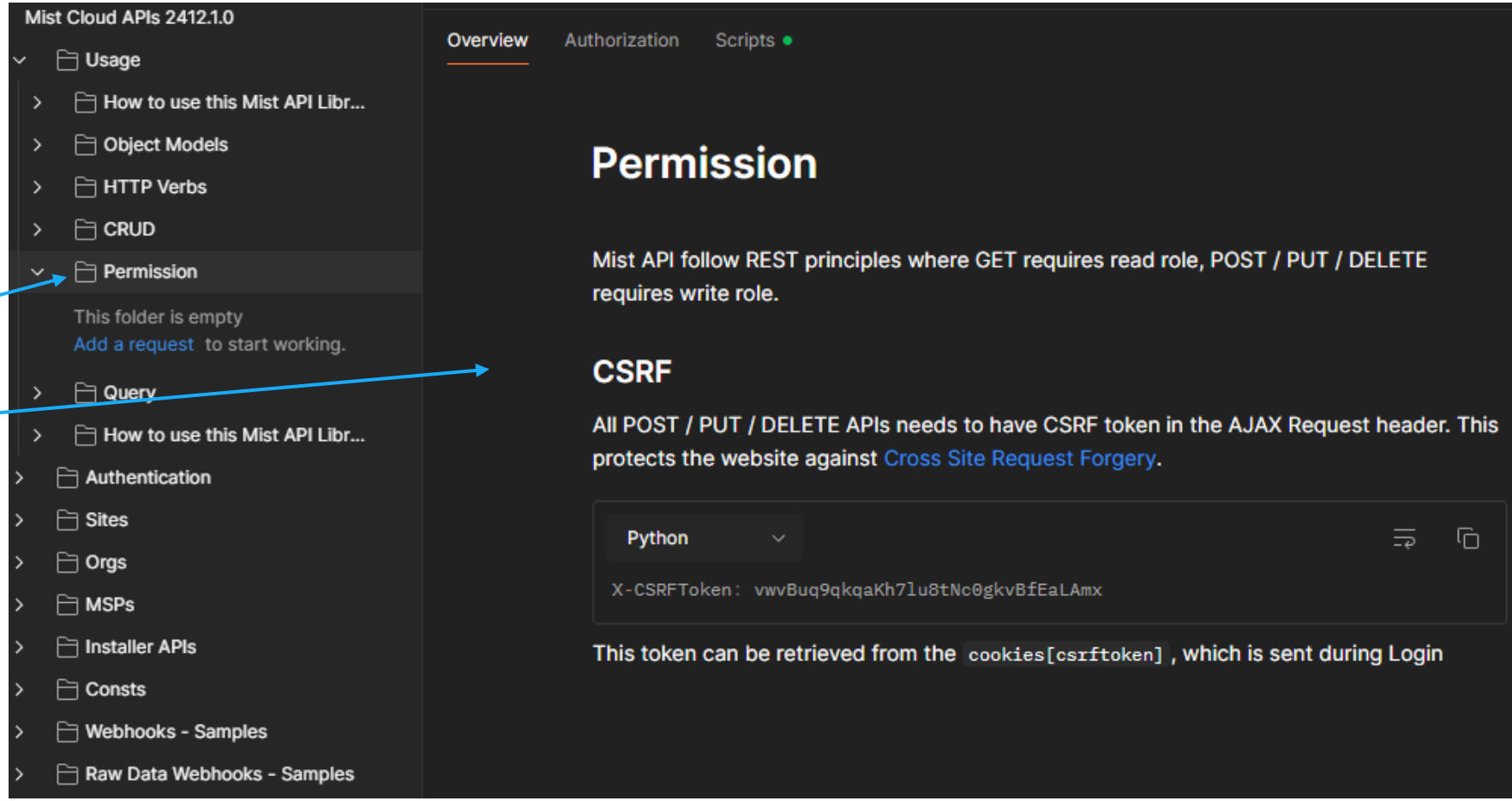
- Select your environment in the environment selector box in the top right corner



- Remember to save your environment by clicking "Save" or hit "Ctrl+S" (Cmd+S)

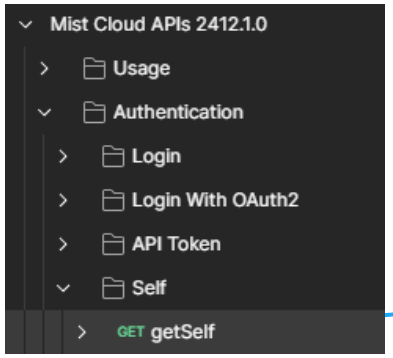
Postman MIST API Collection

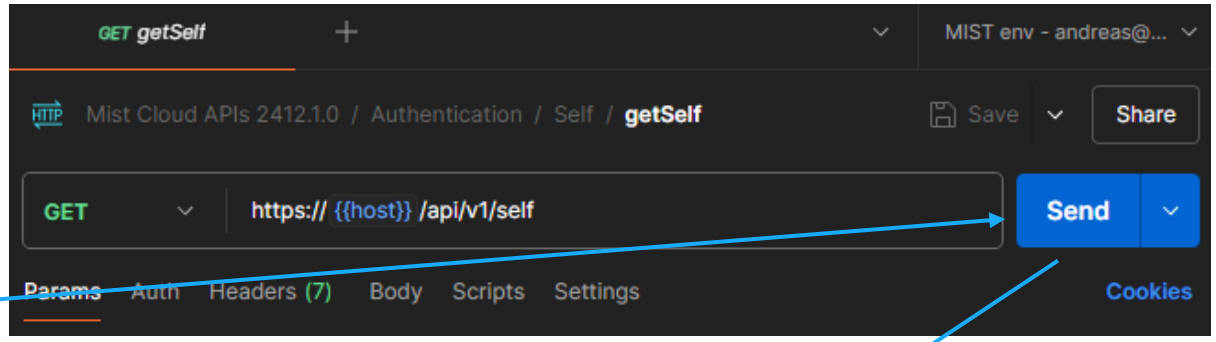
- If you look in the "Usage" folder, there are lots of information
- For example, if you are going to do an operation which require write access, have a look in the "Permission" section here. When you select the "Permission" folder, some info will show in the main window



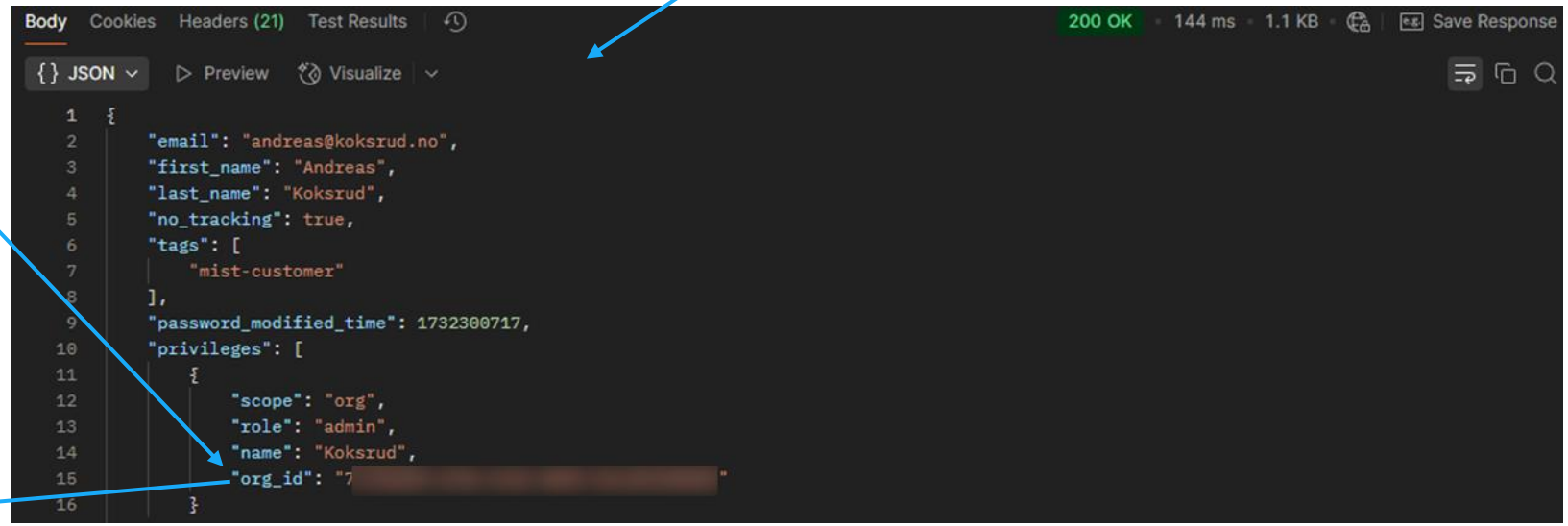
Postman MIST API Collection

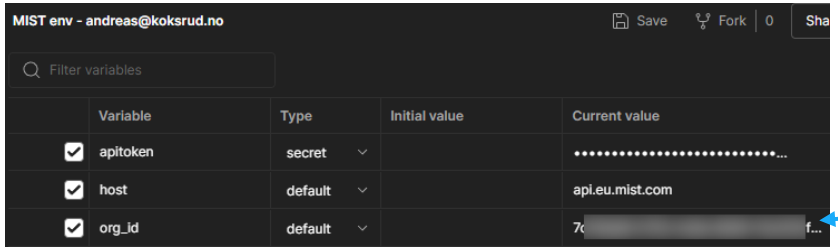
- You can now try out the "getSelf" API call. It will get your user and info about everything you can access





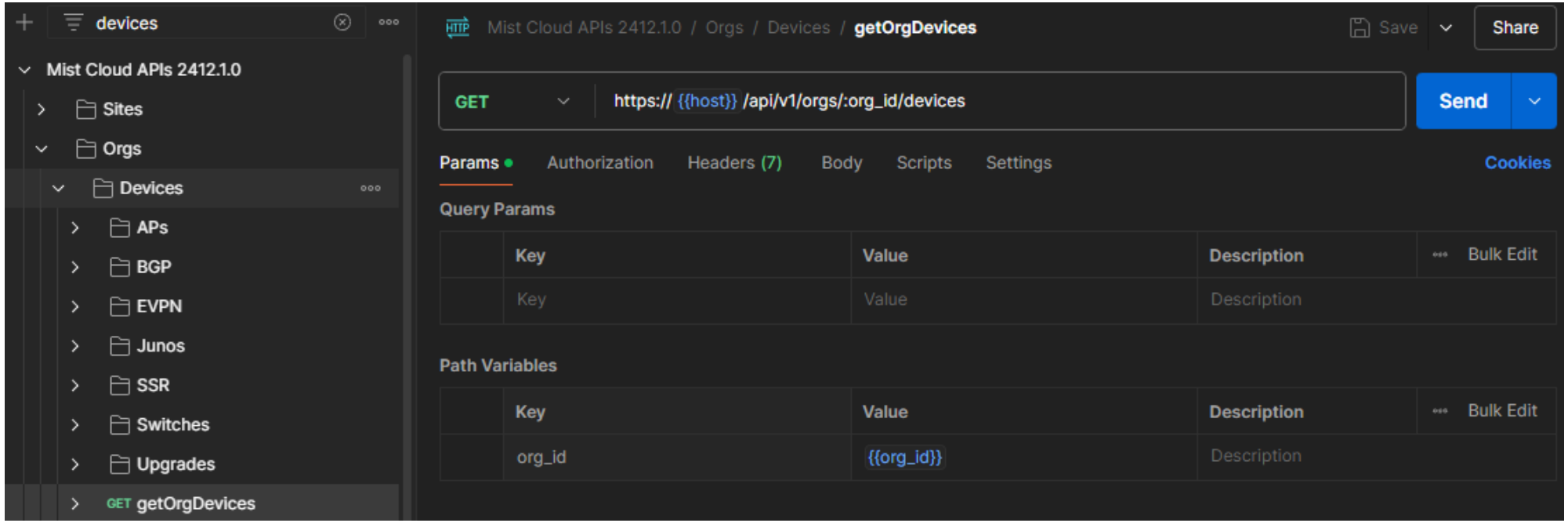
- For further queries to that organization, you can save another environment variable with your organization ID. Call this variable "org_id"

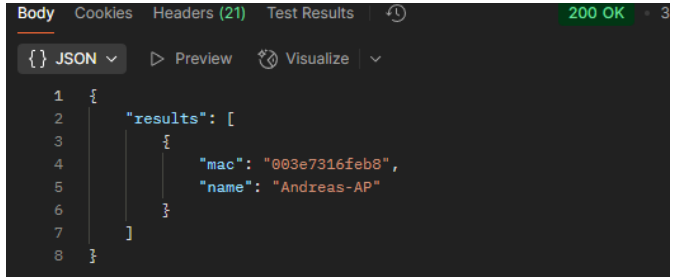




Postman MIST API Collection

- After getting your Org ID, you can get the Devices

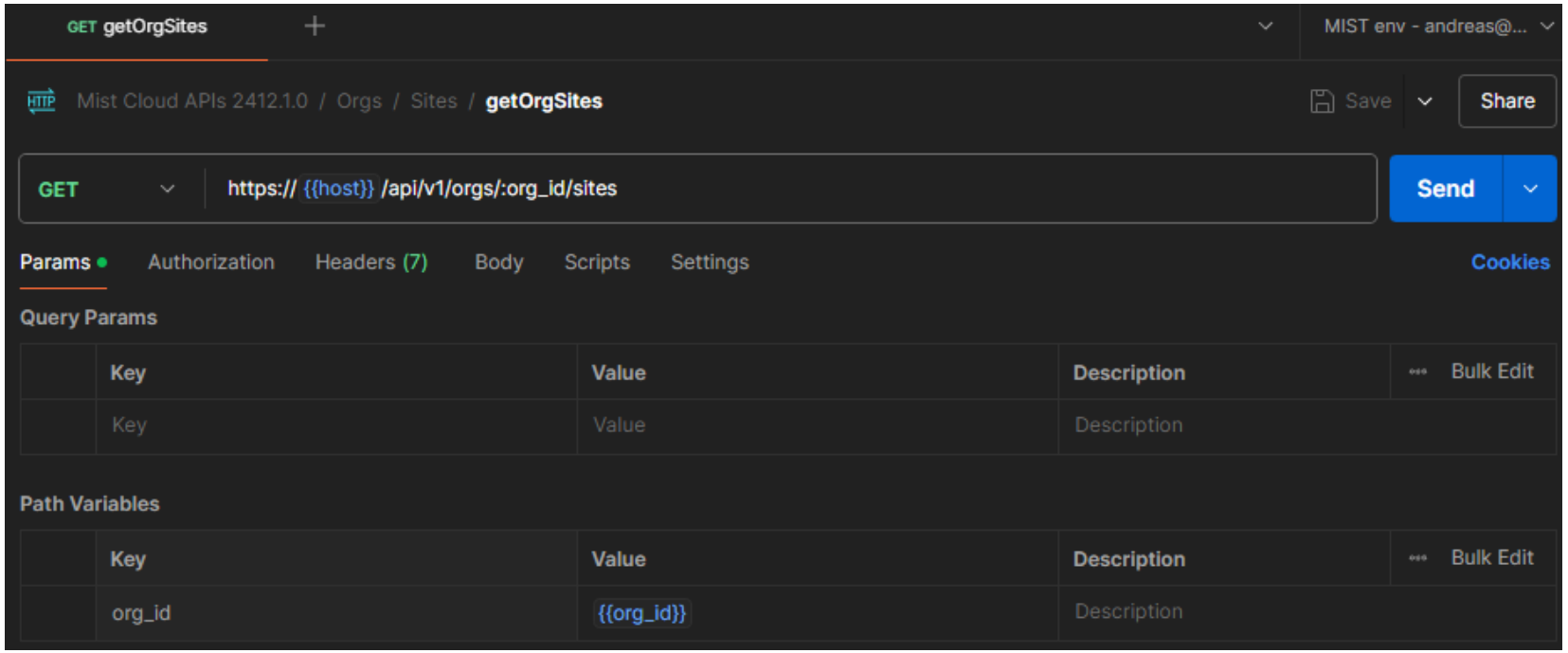




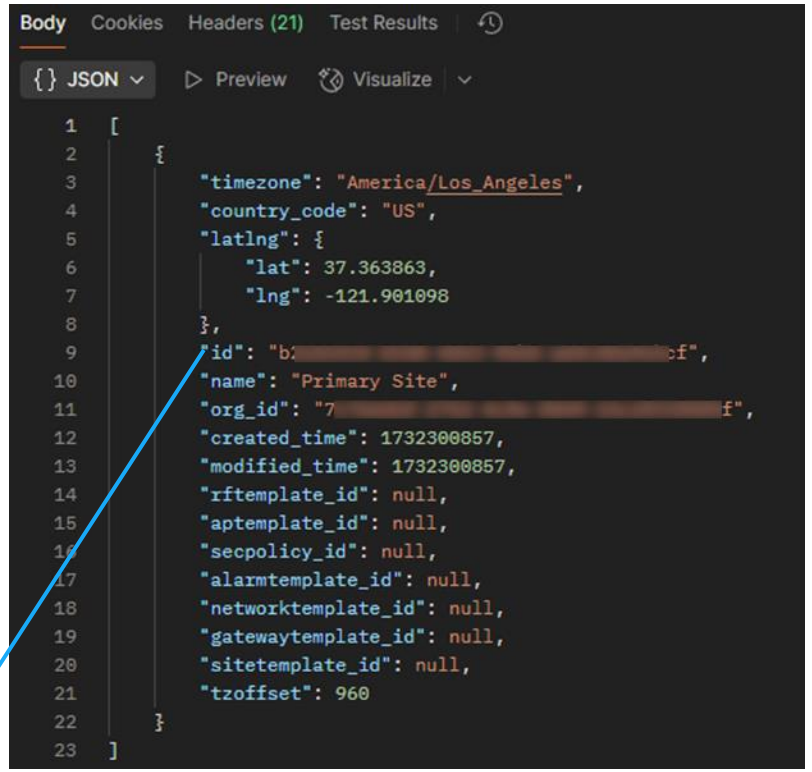
- You can also try "getOrgDeviceStats" for some extended info
- You can also try "getSiteDevices" for some more info on the devices for a site. You need the site_id for this, see next page.

Postman MIST API Collection

- You can also get the Sites in your Org

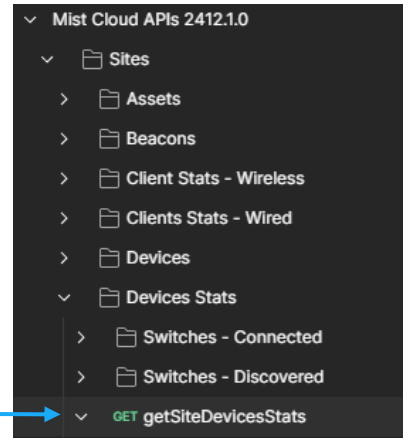


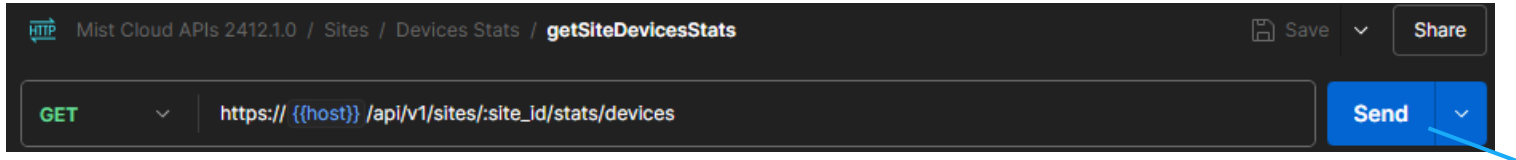
- Put the "id" value in the "site_id" variable in your environment



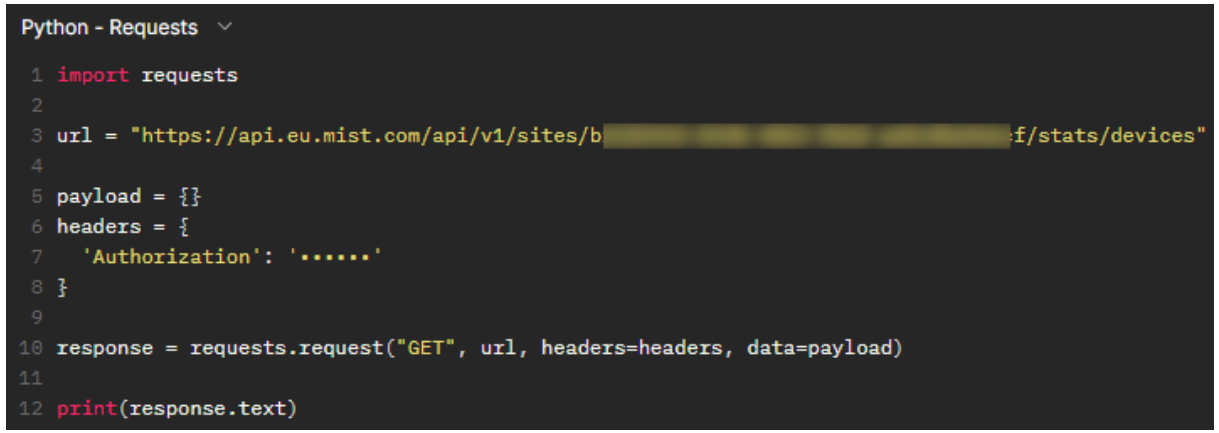
Python - Use the Postman API

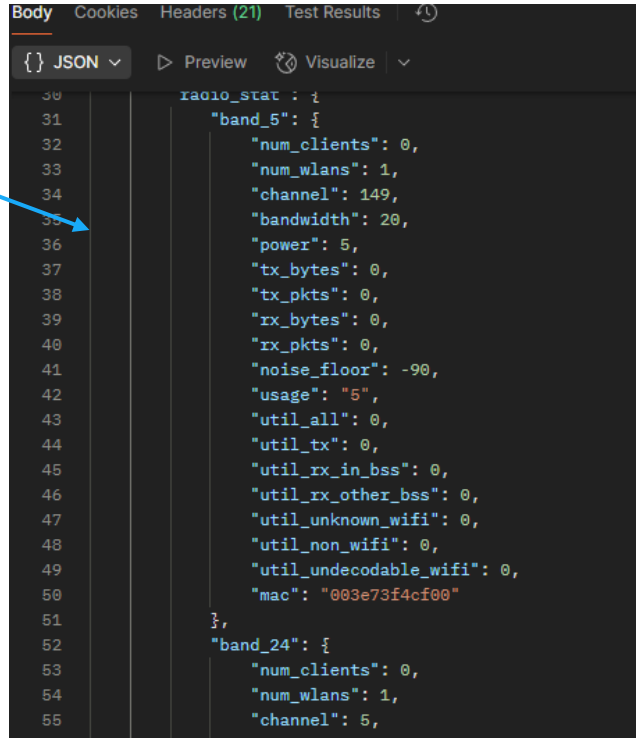
- Try using one of the APIs from the Postman collection. In this example we will get statistics from all devices on a site. For this request to work you will need to have gathered the site_id, and you need to have at least one device in your inventory. Start by running the request in Postman





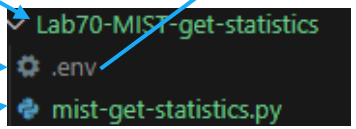
- There is a lot of statistics for each device, like radio statistics
- Then click the "code" button </> and select "Python - Requests" to get an example





Python - Use the MIST API

- Create a new folder "Lab70-MIST-get-statistics"
  - Create a .env file, and fill inn your values from the Postman part of the lab
  - Create an empty file named "mist-get-statistics.py"
- Copy the empty script here to your "mist-get-statistics.py". All functions are just printing a headline
- Remember to save both files ☺
- You can run the script from VS Code or command line



Lab70-MIST-get-statistics > .env

```
1  apitoken='C
2  host='api.eu.mist.com'
3  org_id='7c7
4  site_id='b2
```

.env

```
apitoken='{YOUR TOKEN}'
host='{YOUR API ENDPOINT URL}'
org_id='{YOUR Org ID}'
site_id='{YOUR Site ID}'
```

```
devnet-adm in ubuntu-devnet in wifi-automation on main [X?↓] via v3.12.3 (automation-venv)
❯ cd Lab70-MIST-get-statistics/

devnet-adm in ubuntu-devnet in wifi-automation/Lab70-MIST-get-statistics on main [X?↓] via v3.12.3 (automation-venv)
❯ python3 mist-get-statistics.py

You are connected to organization 7c76ddbf-2752-4c9e-8049-15c2f33f0f8f and network b2183434-8340-4853-95b8-a68c88a9ebcf
Menu:
1. Show device summary
2. Show client summary
3. Show device statistics
4. Quit
Enter choice:
```

mist-get-statistics.py

```
from dotenv import dotenv_values
import os

# Import variables from the .env file
env = dotenv_values('.env')
apitoken = env['apitoken']
host=env['host']
org_id=env['org_id']
site_id=env['site_id']

# Define functions
def show_device_summary():
    print('Device summary:\n--------------------------------------')

def show_client_summary():
    print('Client summary:\n--------------------------------------')

def show_device_statistics():
    print(f"Full Statistics for all devices on site:\n--------------------------------------")

# Text menu with 3 options that run each of the functions, or quit
while True:
    # Clear the screen
    print(f'\nYou are connected to organization {org_id} and site {site_id}')
    print("Menu:")
    print("1. Show device summary")
    print("2. Show client summary")
    print("3. Show device statistics")
    print("4. Quit")

    choice = input("Enter choice: ")
    os.system('cls' if os.name == 'nt' else 'clear')

    if choice == "1":
        show_device_summary()
    elif choice == "2":
        show_client_summary()
    elif choice == "3":
        show_device_statistics()
    elif choice == "4":
        break
    else:
        print("Invalid choice. Please choose again.")
```


Python - Use the MIST API

- Merge the code snippet from Postman into the function show_device_summary()
  - Import the requests package (top of file)
  - Write the rest of the Python code into the show_device_summary() function
  - We change the url with an f-string and put in the variable "site_id" from the .env file instead
  - The Token value pasted from Postman is the cleartext of your token. Change this to an f-string as well, and insert the "apitoken" from the .env file

```python
import requests

def show_device_summary():
    print('Device summary:\n-----------------------------------')

    url = f"https://api.eu.mist.com/api/v1/sites/{site_id}/devices"

    payload = {}
    headers = {
    'Authorization': f'Token {apitoken}'
    }

    response = requests.request("GET", url, headers=headers, data=payload)

    print(response.text)
```

Postman:

- Run your script and try menu choice 1.
- The output should be there, but it is not very nice looking right now. To make this easier to read, import the "pprint" module in the top of your script, and print using "pprint" instead of "print"

```
3   from pprint import pprint
```

```
25
26      pprint(response.json())
27
```


Python - Use the MIST API

- Do the same for show_client_summary()
  Use searchSiteWirelessClients from Postman
  Change the hardcoded Site ID from Postman
  with {site_id} from .env.
  Use pprint for printing.

```
def show_client_summary():
    print('Client summary:\n-----------------------------------')
    url = f"https://api.eu.mist.com/api/v1/sites/{site_id}/clients/search"

    payload = {}
    headers = {
    'Authorization': f'Token {apitoken}'
    }

    response = requests.request("GET", url, headers=headers, data=payload)

    pprint(response.json())
```

- Do the same for show_device_statistics()
  Use getSiteDevicesStats from Postman
  Change the hardcoded Site ID from Postman
  with {site_id} from .env.
  Use pprint for printing.

```
def show_device_statistics():
    print(f"Full Statistics for all devices on site:\n----------------------
    url = f"https://api.eu.mist.com/api/v1/sites/{site_id}/stats/devices"

    payload = {}
    headers = {
    'Authorization': f'Token {apitoken}'
    }

    response = requests.request("GET", url, headers=headers, data=payload)

    pprint(response.json())
```

- Remember to save the file before running the script ("python3 mist-get-statistics.py")

Python - Use the MIST API

- Some suggestions for next steps could be
  - Save the JSON to a JSON file (use the method from Lab 40/44 where you saved run-config)
  - Flatten the JSON output to a pandas dataframe and save to CSV or Excel (From Lab 41)

Lab Exercise #71: Python - MIST->InfluxDB->Grafana

- This lab exercise will be the preparation for a Grafana dashboard visualizing data from our MIST APs.We will write a simple Python script that
  1. Pulls data from MIST APs
  2. Inserts the data in InfluxDB
- The Grafana dashboard itself will be vendor independent, the same dashboard as is used in the Meraki exercise (Lab exercise #61)
- In this exercise we will build on the previous exercise by using the RESTCONF calls that we can figure out by using the MIST API collection in Postman. You will use the following RESTCONF call in this exercise
  - getSiteDevicesStats
    (MIST Cound APIs v2412.1.0 > Sites > Devices Stats > getSiteDevicesStats)

Prerequisites - Python packages

- To prepare we will install some Python packages in our "automation-venv" virtual environment
  - InfluxDB package
    - References: https://github.com/influxdata/influxdb-python
    - References: https://influxdb-python.readthedocs.io/en/latest/
- Start by creating the Lab Exercise folder

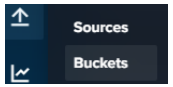
```
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/$ cd ~
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/$ mkdir Lab71_Python-MIST-InfluxDB
```

- Then ensure you are in the automation-venv, and use pip to install influxdb

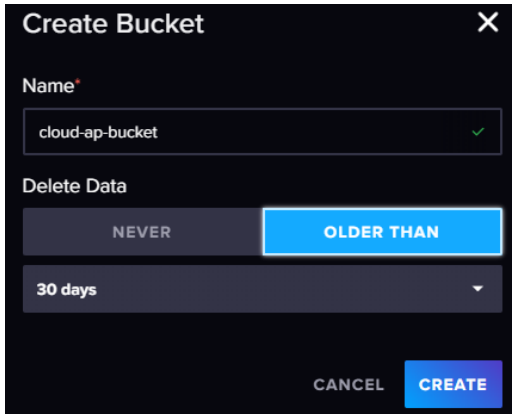
```
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab71_Python-MIST-InfluxDB$ source ~/automation-venv/bin/activate
(automation-venv) devnet-adm@ubuntu-devnet:~/wifi-automation/Lab71_Python-MIST-InfluxDB$ pip install influxdb
```


Prerequisites - InfluxDB databases

- We will also create a new bucket + database in InfluxDB
- Login to InfluxDB - http://{ubuntu-devnet}:8086/
- Go to Buckets



- Create a new bucket "cloud-ap-bucket"

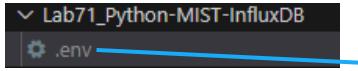


- Create the database for this bucket (so we can use it with InfluxQL)

```
devnet-adm@ubuntu-devnet:~/tig-stack$ docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED       STATUS       PORTS
56c9135dbb27   influxdb:2.7.10        "/entrypoint.sh infl…"   5 hours ago   Up 5 hours   0.0.0.0:8086->8086/tcp, :::8086->8086/tcp
devnet-adm@ubuntu-devnet:~/tig-stack$ docker exec -it <your_container-id> bash
root@56c9135dbb27:/# influx v1 dbrp list --org-id <your_org-id> --token <your_admin_token>
ID      Database        Bucket ID       Retention Policy        Default Organization ID
(…)
cf94ebbf115524c3        cloud-ap-bucket cf94ebbf115524c3        autogen                 true    f8f6ee5d500865de

root@56c9135dbb27:/# influx v1 dbrp create --db cloud-ap-db --rp cloud-ap-rp --bucket-id cf94ebbf115524c3 --default --org-id <your_org-id> --token <your_admin_token>

ID                      Database        Bucket ID               Retention Policy        Default Organization ID
0df9b97891720000        cloud-ap-db     cf94ebbf115524c3        cloud-ap-rp             true    f8f6ee5d500865de

root@56c9135dbb27:/# exit
devnet-adm@ubuntu-devnet:~/tig-stack$
```

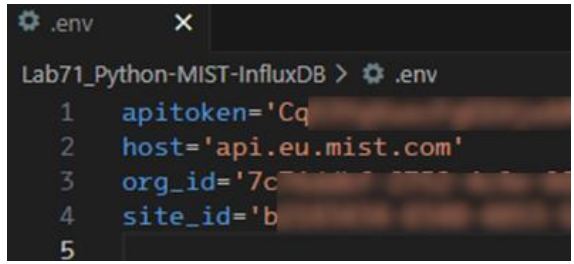

Pulling MIST data using Python

- Now we will create the Python script that pulls data from the MIST API, and inserts this into our influxdb database
- Start by creating the .env file. You can copy this from the previous exercise.
- Insert your apitoken, host, org_id and site_id into the file if you did not have them from last exercise
- !! This script assumes that you have one or more MIST APs in your organization !!



.env

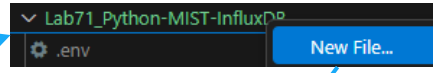
```
apitoken='{YOUR TOKEN}'
host='{YOUR API ENDPOINT URL}'
org_id='{YOUR Org ID}'
site_id='{YOUR Site ID}'
```

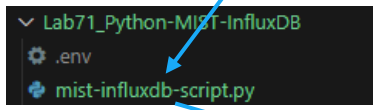


Pulling MIST data using Python

- Create the Python file "mist-influxdb-script.py"
- In this exercise you can copy-paste the example to the right, and we will explain it part by part in the next slides
- Output should be something like this

```
devnet-adm in ubuntu-devnet in wifi-automation/Lab71_Python-MIST-InfluxDB on main [?] via
❯ python3 mist-influxdb-script.py
Updated InfluxDB with data from Andreas-AP at 2025-02-01T17:16:00Z
Updated InfluxDB with data from Andreas-AP at 2025-02-01T17:16:10Z
Updated InfluxDB with data from Andreas-AP at 2025-02-01T17:16:20Z
Updated InfluxDB with data from Andreas-AP at 2025-02-01T17:16:30Z
Updated InfluxDB with data from Andreas-AP at 2025-02-01T17:16:41Z
Updated InfluxDB with data from Andreas-AP at 2025-02-01T17:16:51Z
Updated InfluxDB with data from Andreas-AP at 2025-02-01T17:17:01Z
Updated InfluxDB with data from Andreas-AP at 2025-02-01T17:17:11Z
```





.env

```
apitoken='{YOUR TOKEN}'
host='{YOUR API ENDPOINT URL}'
org_id='{YOUR Org ID}'
site_id='{YOUR Site ID}'
```

mist-influxdb-script.py

```
from influxdb import InfluxDBClient
from datetime import datetime, timezone
from dotenv import dotenv_values
import requests
import json
import time

# Import variables from the .env file
env = dotenv_values('.env')
host = env['host']
apitoken = env['apitoken']
org_id=env['org_id']
site_id=env['site_id']

def get_device_statistics():
    url = f"https://{host}/api/v1/sites/{site_id}/stats/devices"

    payload = {}
    headers = {
    'Authorization': f'Token {apitoken}'
    }

    response = requests.request("GET", url, headers=headers, data=payload)
    return response.json()

def update_influxDB(measurement_name, hostname, fields):
    timestamp = datetime.now(timezone.utc).strftime('%Y-%m-%dT%H:%M:%SZ')
    json_body = [
        {
            "measurement": measurement_name,
            "tags": {
                "host": hostname,
            },
            "time": timestamp,
            "fields": fields
        }
    ]
    client = InfluxDBClient('localhost', 8086, 'devnet-adm', 'ChangeMe2025!', 'cloud-ap-db')
    client.write_points(json_body)
    print(f"Updated InfluxDB with data from {hostname} at {timestamp}")

# Iterate through all devices and do stuff
def iterate_access_points():
    devices = get_device_statistics()
    for device in devices:
        if device['type'] == "ap":
            interference = {}
            for band in device['radio_stat']:
                interference[band+"GHz Wi-Fi"] = device['radio_stat'][band]['util_rx_in_bss'] +
device['radio_stat'][band]['util_rx_other_bss'] + device['radio_stat'][band]['util_unknown_wifi'] +
device['radio_stat'][band]['util_undecodable_wifi']
                interference[band+"GHz non-Wi-Fi"] = device['radio_stat'][band]['util_non_wifi']
                interference[band+"GHz Total"] = device['radio_stat'][band]['util_all']
            update_influxDB('ch_utilization', device['name'], interference)

while True:
    iterate_access_points()
    time.sleep(10)
```

Pulling MIST data using Python

- Imports at the start of the file
  - influxdb module, to communicate with InfluxDB
  - datetime, to timestamp our values
  - dotenv to import values from the .env file
  - requests to use REST APIs
  - time, we use this to pause the script for 10 seconds before we poll the data once more
- Import the .env file variables

solutions > Lab71_Python-MIST-InfluxDB > mist-influxdb-script.py > ...

```
1  from influxdb import InfluxDBClient
2  from datetime import datetime, timezone
3  from dotenv import dotenv_values
4  import requests
5  import time
6
7  # Import variables from the .env file
8  env = dotenv_values('.env')
9  host = env['host']
10 apitoken = env['apitoken']
11 org_id=env['org_id']
12 site_id=env['site_id']
```


Pulling MIST data using Python

- get_device_statistics() function
  - URL is created using the host and site_id from you env variables (imported from .env file)
  - Authorization header created using your apitoken variable (imported from the .env file)
  - A request is made, using your URL and the Authorization token in the headers
  - The "payload" is empty. It can be for example when writing stuff, or for getting stuff where you can search/filter by certain parameters
  - JSON representation of the response is returned to wherever you called the function from
- update_influxDB() function
  - Function to write values to your local influxDB
  - We create a timestamp to go with the measurement
  - We will get measurement_name, hostname and fields from the get_channel_utilization function and use them when we call this function
  - Create the InfluxDBClient object, which we will use this to write to influxdb. In this lab exercise it points to "localhost" since the influxDB is running on the same host. We could also use the IP of our InfluxDB server here.

```python
def get_device_statistics():
    url = f"https://{host}/api/v1/sites/{site_id}/stats/devices"

    payload = {}
    headers = {
    'Authorization': f'Token {apitoken}'
    }

    response = requests.request("GET", url, headers=headers, data=payload)
    return response.json()

def update_influxDB(measurement_name, hostname, fields):
    timestamp = datetime.now(timezone.utc).strftime('%Y-%m-%dT%H:%M:%SZ')
    json_body = [
        {
            "measurement": measurement_name,
            "tags": {
                "host": hostname,
            },
            "time": timestamp,
            "fields": fields
        }
    ]
    client = InfluxDBClient('localhost', 8086, 'devnet-adm', 'ChangeMe2025
    client.write_points(json_body)
    print(f"Updated InfluxDB with data from {hostname} at {timestamp}")
```


Pulling MIST data using Python

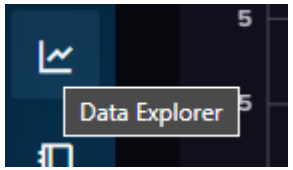
- iterate_access_points() function
  - Get your devices using the get_device_statistics() function
  - Iterate over all devices in the "devices" list
    Start by checking if the device type is "ap", we do not want do do anything with other devices in this script
  - We create an empty dictionary "interference", and we populate this with the statistics. The "bandname" stuff is not strictly neccessary, but since we will use this in a cross-platform dashboard in Grafana, we want the keys to have identical names regardless of vendor
  - Update InfluxDB for each device

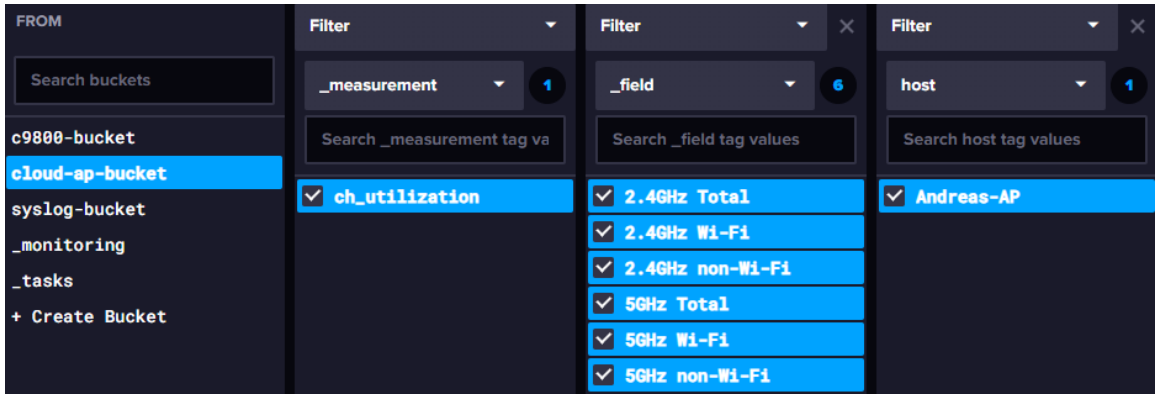
```
41  # Iterate through all devices and do stuff
42  def iterate_access_points():
43      devices = get_device_statistics()
44      for device in devices:
45          if device['type'] == "ap":
46              interference = {}
47              for band in device['radio_stat']:
48                  if band=="band_24": bandname = "2.4"
49                  if band=="band_5": bandname = "5"
50                  if band=="band_6": bandname = "6"
51                  interference[bandname+"GHz Wi-Fi"] = device['radio_stat'][band]['util_rx_in_
52                  interference[bandname+"GHz non-Wi-Fi"] = device['radio_stat'][band]['util_no
53                  interference[bandname+"GHz Total"] = device['radio_stat'][band]['util_all']
54              update_influxDB('ch_utilization', device['name'], interference)
55
56  while True:
57      iterate_access_points()
58      time.sleep(10)
```

- Use a while loop, that loops forever (until you Ctrl+Z), and sleep for 10 seconds each time you have run the iterate_access_points() function

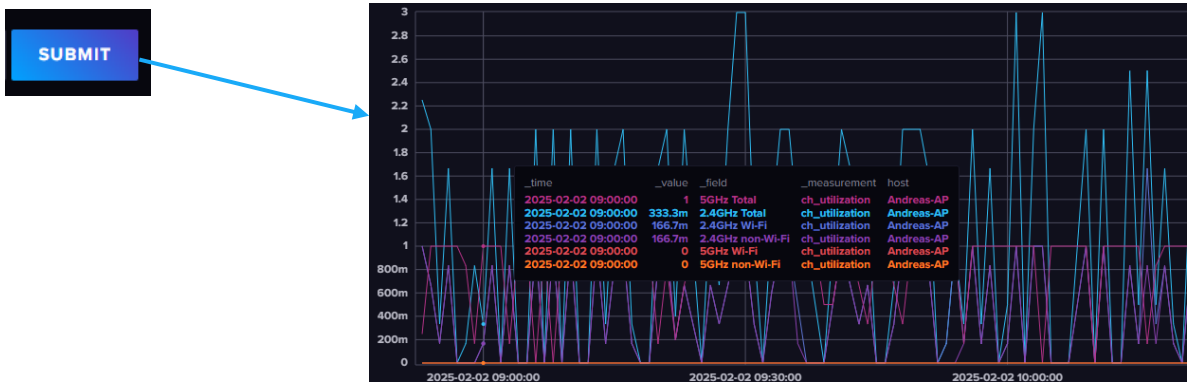
Checking the data in InfluxDB

- Log in to influxdb at http://{UBUNTU_IP}:8086
- Go to Data Explorer, choose cloud-ap-bucket and verify if there is some populated data there



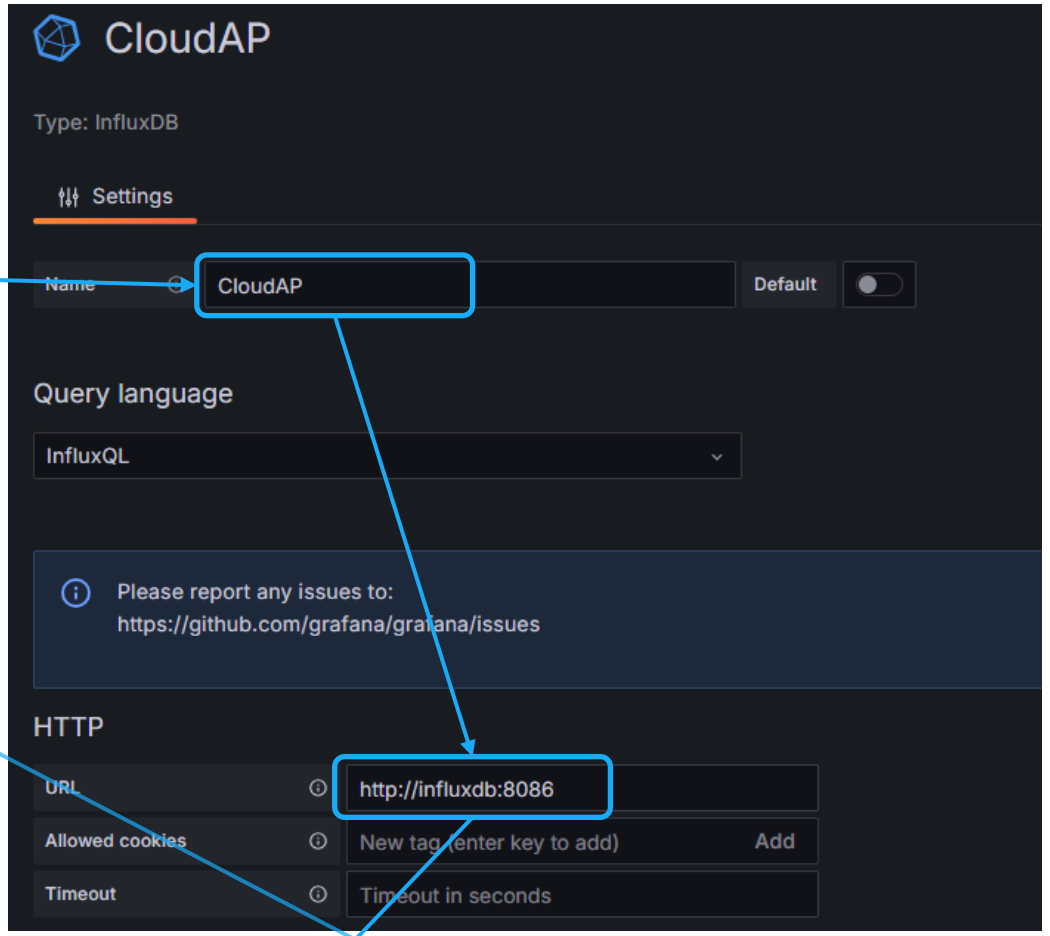


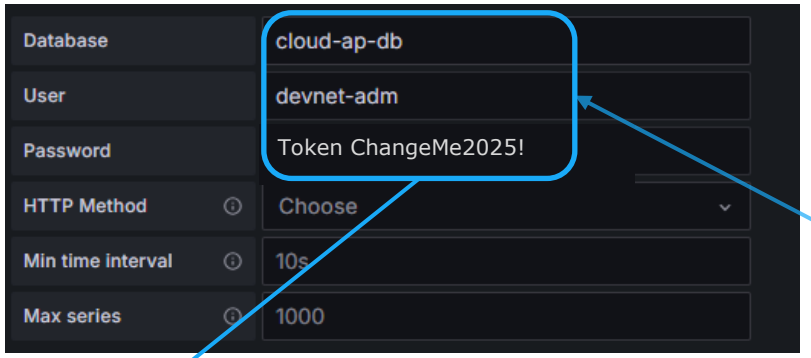
- Submit and see if the graph shows some values



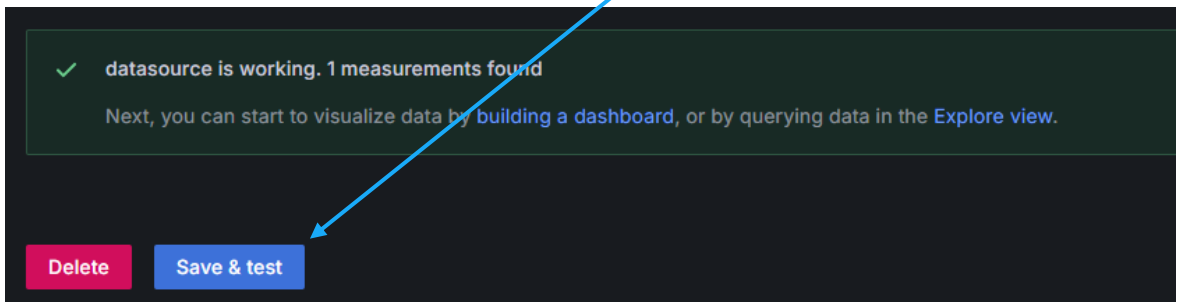
Creating Datasource in Grafana

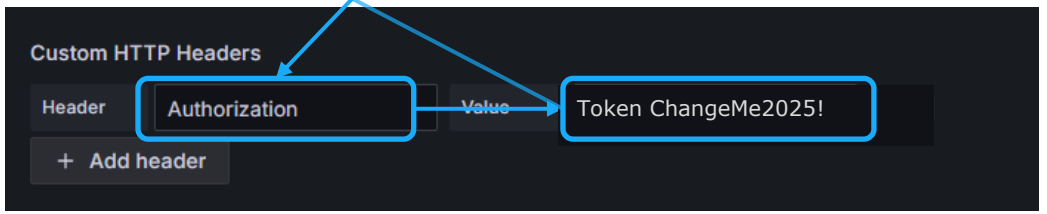
- Login to Grafana - http://{ubuntu-devnet}:3000/
- Create a new datasource > InfluxDB
- The reason we create a new datasource, is to select a different database than the one we used for the 9800 telemetry





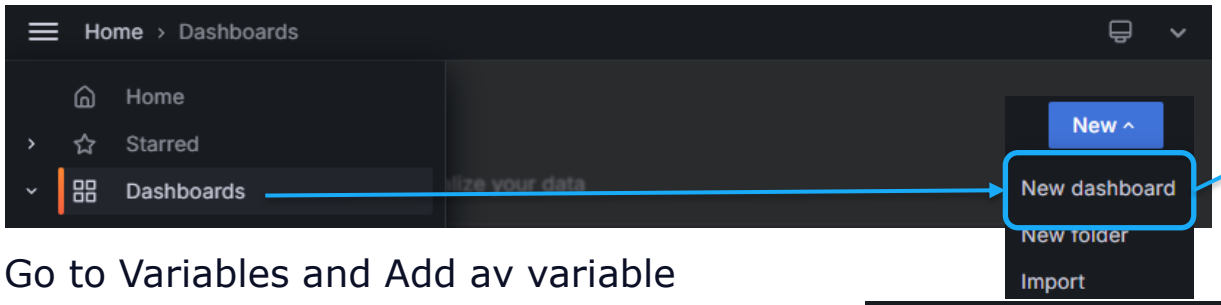
- "Save & test" should give 1 measurement

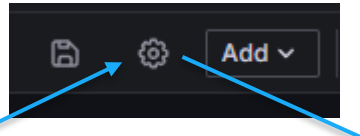


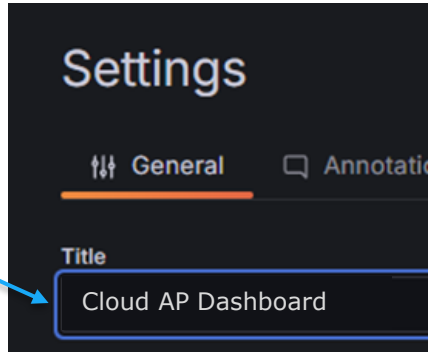


Creating the dashboard in Grafana

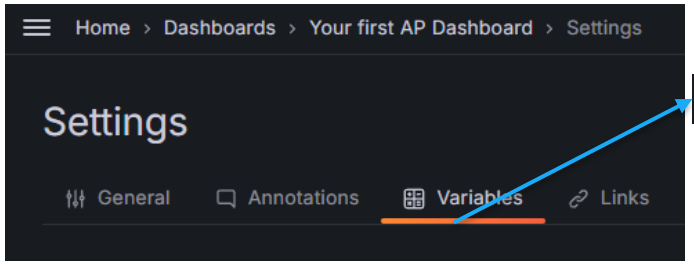
- Create a new Dashboard. Name it "Cloud AP Dashboard" or something

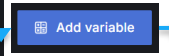


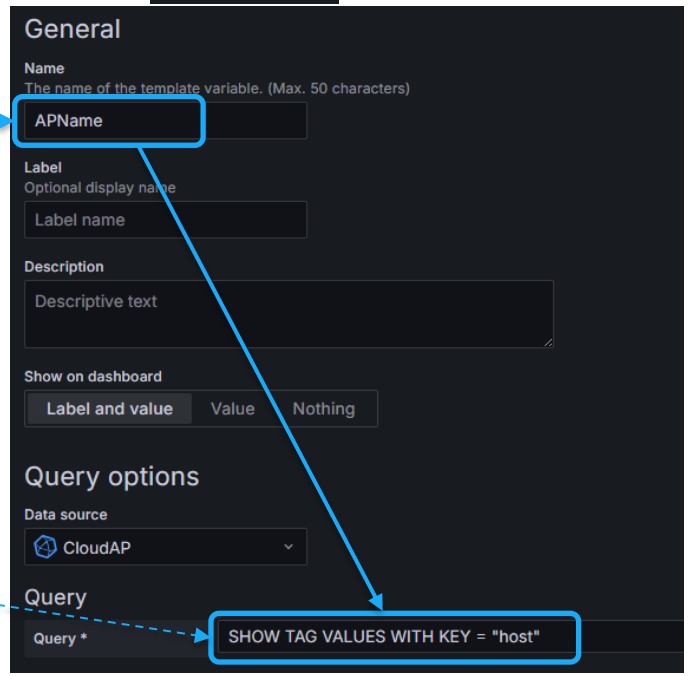




- Go to Variables and Add av variable



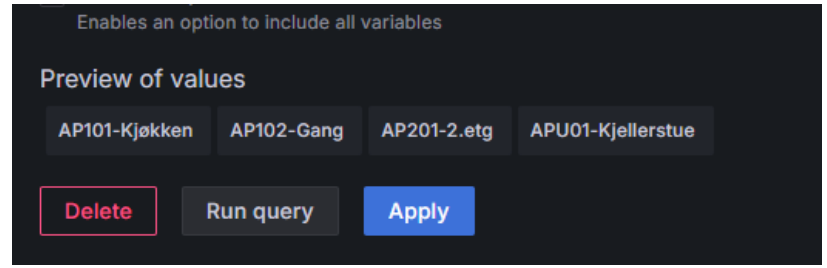




- We show values with key "host". The reason for this is that we put data in influxDB with that key

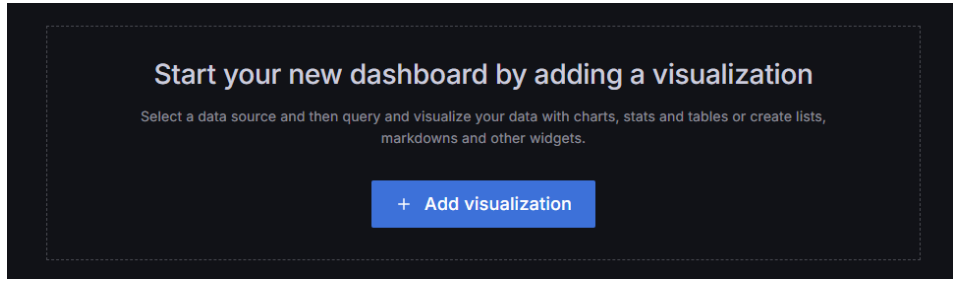
```
json_body = [
    {
        "measurement": measurement_name,
        "tags": {
            "host": hostname,
        },
        "time": timestamp,
        "fields": fields
    }
]
```

- When you press "Run query" you should see the names of your APs that you have inserted into influxdb

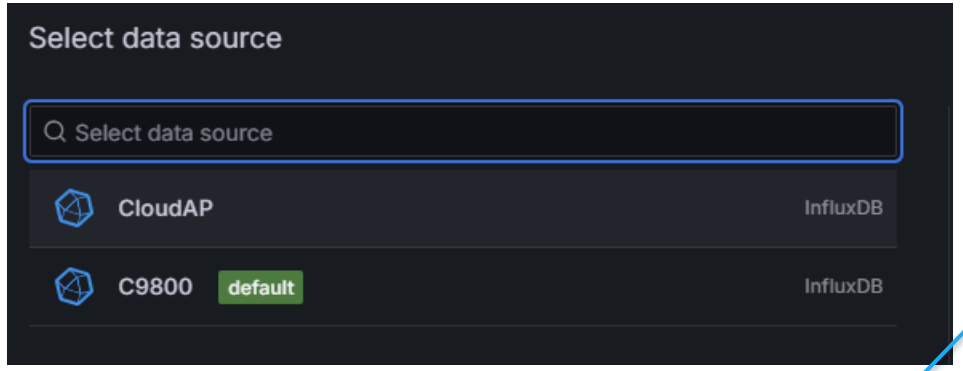


Creating the dashboard in Grafana

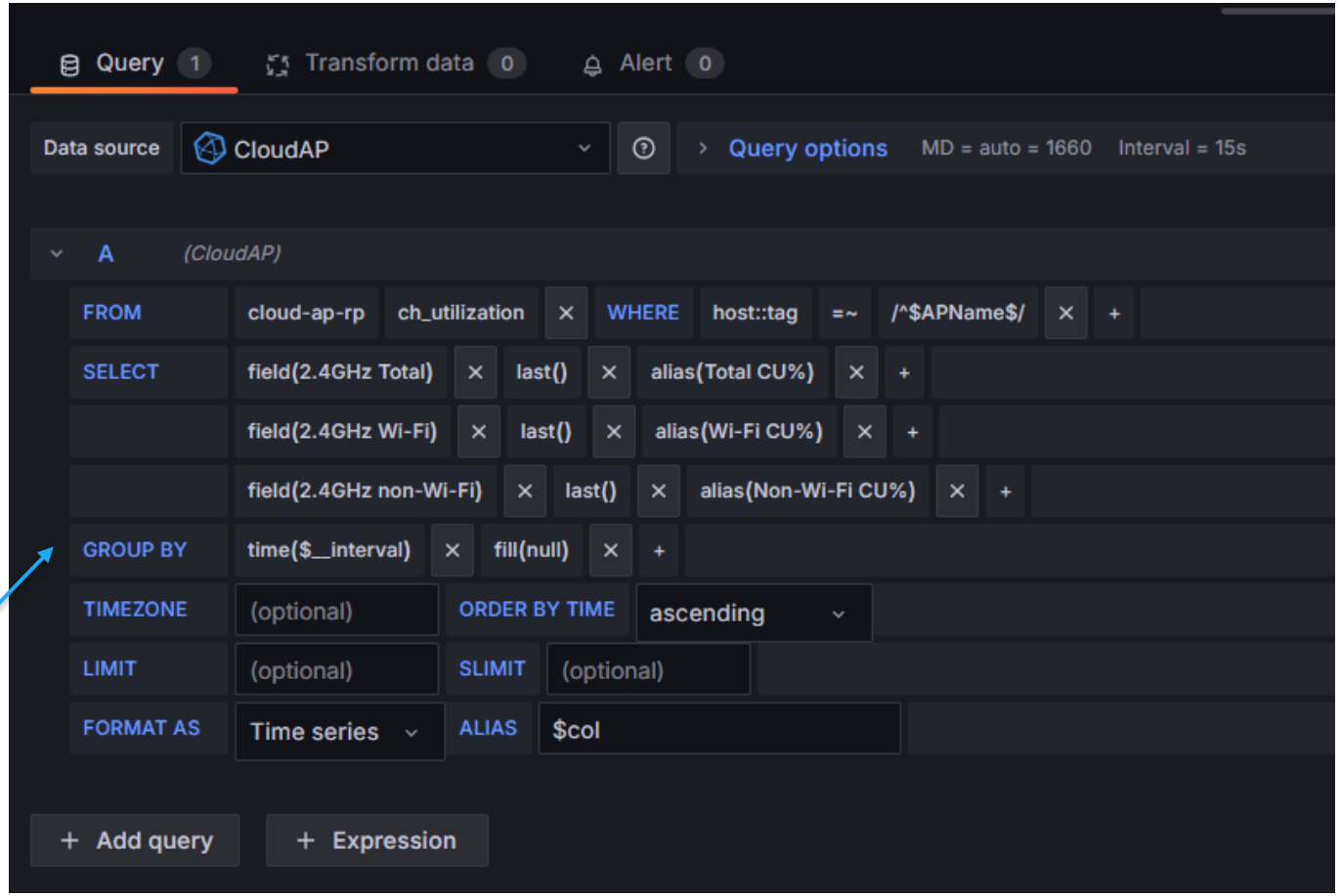
- Go to the dashboard and select "Add visualization"



- Select the "CloudAP" datasource that you created

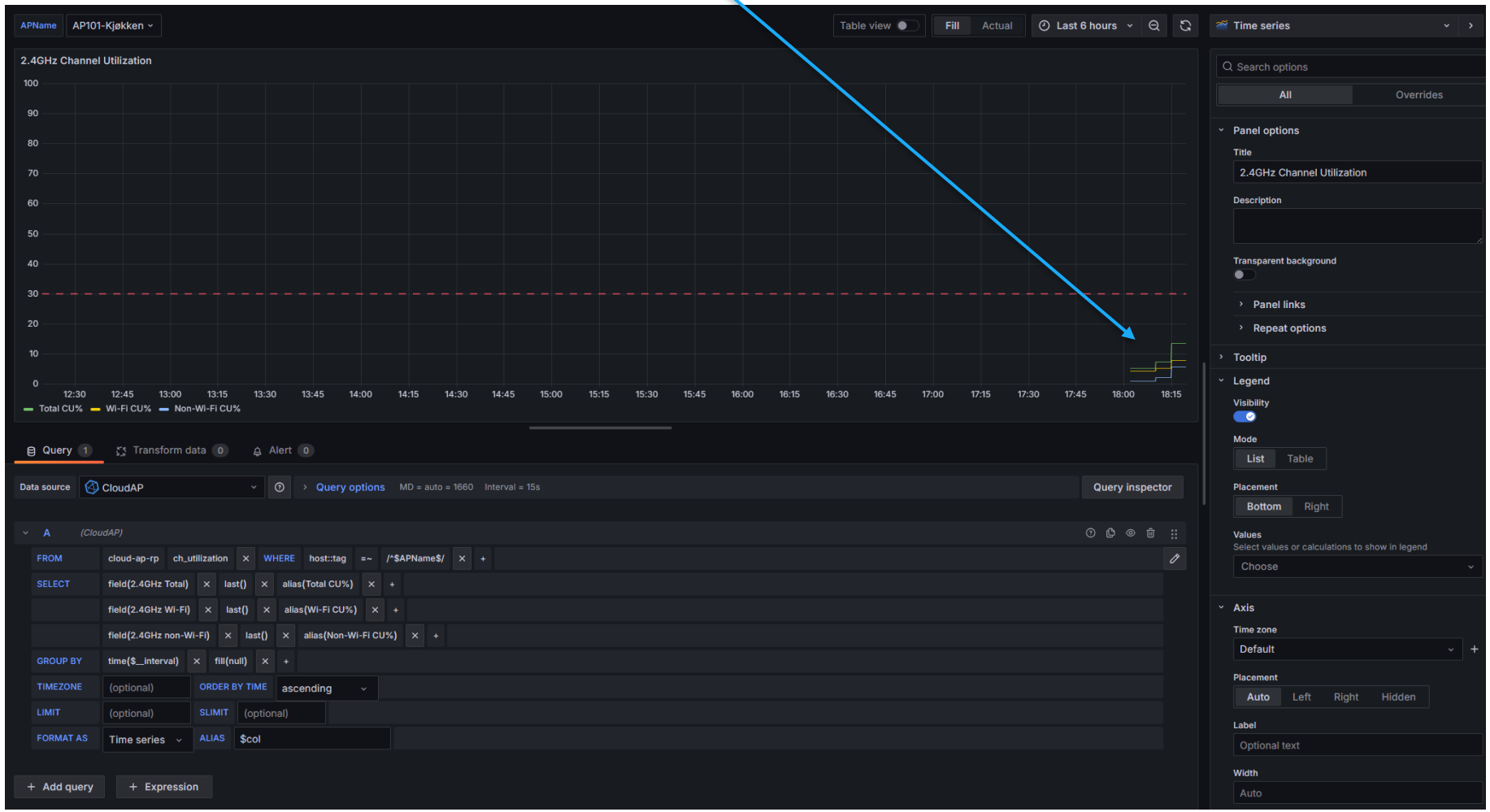


- Replicate this query for your first visualization



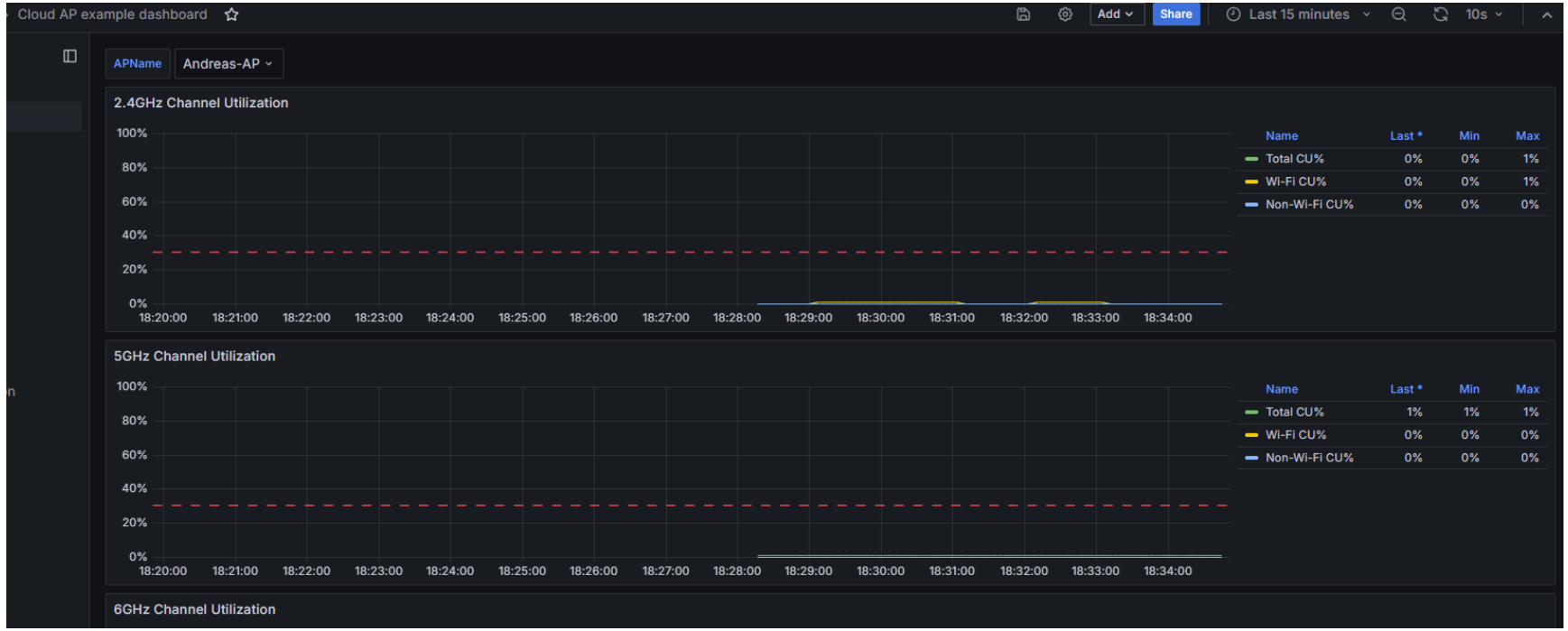
Creating the dashboard in Grafana

- You should see the graphs starting to show



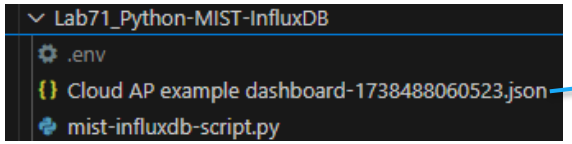
Example dashboard in Grafana

- Duplicate the 2.4GHz visualization and create a panel for 5GHz and 6GHz

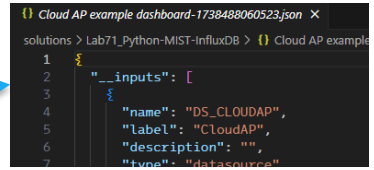


- You can import the example dashboard from the Lab71 folder in "solutions"

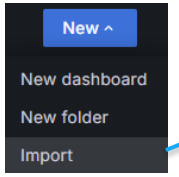
VS Code, open the JSON file



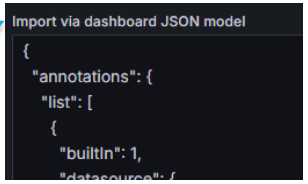
Copy the text



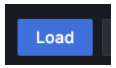
Grafana (Dashboards)



Paste the JSON



press "Load"



Lab Exercise #72: Expanding the dashboard

- These are all in the getSiteDevicesStats which we already pull in the Python script. We will use them to build a couple of panels to expand our Dashboard
  - Channel history panel
    Client count: `radio_stat → band_5 → num_clients`
    Current channel: `radio_stat → band_5 → channel`
    Channel width: `radio_stat → band_5 → bandwidth`
    TxPower: `radio_stat → band_5 → power`
    Number of clients: `radio_stat → band_5 → num_clients`
  - AP info panel
    AP Name: `already saved in the $APName variable in Grafana`
    AP Model: `model`
    IP Address: `ip`
    External IP: `ext_ip`
    Mount direction: `mount`
    Power source: `power_src`
    Ethernet MAC: `mac`
    Radio MAC 5GHz: `radio_stat → band_5 → mac`
  - For each new value you will need to modify the Python file to insert data into InfluxDB. You can then pull the data into your Grafana Dashboard. The next slides will give examples and explanations on this for the above values

.env

```
apitoken='{YOUR TOKEN}'
host='{YOUR API ENDPOINT URL}'
org_id='{YOUR Org ID}'
site_id='{YOUR Site ID}'
```

mist-influxdb-script-expanded.py

```
from influxdb import InfluxDBClient
from datetime import datetime, timezone
from dotenv import dotenv_values
import requests
import time

# Import variables from the .env file
env = dotenv_values('.env')
host = env['host']
apitoken = env['apitoken']
org_id=env['org_id']
site_id=env['site_id']

def get_device_statistics():
    url = f"https://{host}/api/v1/sites/{site_id}/stats/devices"

    payload = {}
    headers = {
    'Authorization': f'Token {apitoken}'
    }

    response = requests.request("GET", url, headers=headers, data=payload)
    return response.json()

def update_influxDB(measurement_name, hostname, fields):
    timestamp = datetime.now(timezone.utc).strftime('%Y-%m-%dT%H:%M:%SZ')
    json_body = [
        {
            "measurement": measurement_name,
            "tags": {
                "host": hostname,
            },
            "time": timestamp,
            "fields": fields
        }
    ]
    client = InfluxDBClient('localhost', 8086, 'devnet-adm', 'ChangeMe2025!', 'cloud-ap-db')
    client.write_points(json_body)
    print(f"Updated InfluxDB with data from {hostname} at {timestamp}")

# Iterate through all devices and do stuff
def iterate_access_points():
    devices = get_device_statistics()
    for device in devices:
        if device['type'] == "ap":
            interference = {}
            for band in device['radio_stat']:
                if band=="band_24": bandname = "2.4"
                if band=="band_5": bandname = "5"
                if band=="band_6": bandname = "6"
                interference[bandname+"GHz Wi-Fi"] = device['radio_stat'][band]['util_rx_in_bss'] +
device['radio_stat'][band]['util_rx_other_bss'] + device['radio_stat'][band]['util_unknown_wifi'] +
device['radio_stat'][band]['util_undecodable_wifi']
                interference[bandname+"GHz non-Wi-Fi"] = device['radio_stat'][band]['util_non_wifi']
                interference[bandname+"GHz Total"] = device['radio_stat'][band]['util_all']
            update_influxDB('ch_utilization', device['name'], interference)

            channel_history = {}
            for band in device['radio_stat']:
                if band=="band_24": bandname = "2.4"
                if band=="band_5": bandname = "5"
                if band=="band_6": bandname = "6"
                channel_history[bandname+"GHz #Clients"] = device['radio_stat'][band]['num_clients']
                channel_history[bandname+"GHz Channel"] = device['radio_stat'][band]['channel']
                channel_history[bandname+"GHz Bandwidth"] = device['radio_stat'][band]['bandwidth']
                channel_history[bandname+"GHz TxPower"] = device['radio_stat'][band]['power']
            update_influxDB('channel_history', device['name'], channel_history)

            device_info = {}
            device_info['model'] = device['model']
            device_info['ip'] = device['ip']
            device_info['ext_ip'] = device['ext_ip']
            device_info['mount'] = device['mount']
            device_info['power_src'] = device['power_src']
            device_info['mac'] = device['mac']
            device_info['radio_mac'] = device['radio_stat']['band_5']['mac']
            update_influxDB('device_info', device['name'], device_info)


while True:
    iterate_access_points()
    time.sleep(10)
```

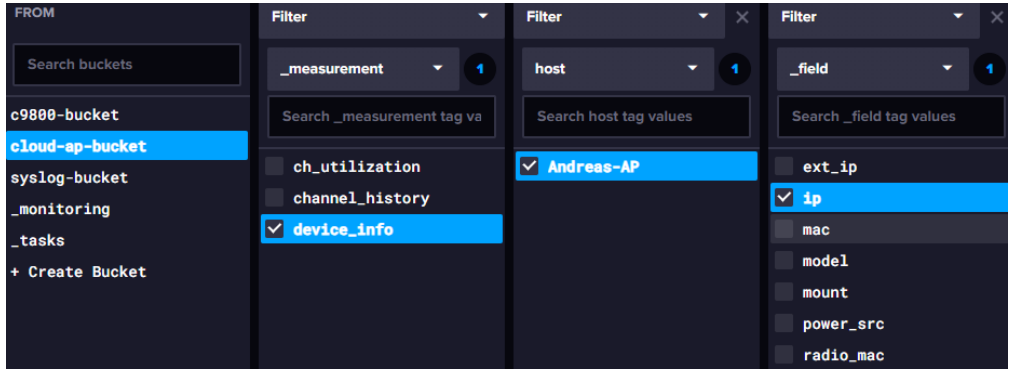

Modifying the Python script

- For this example (and the solutions folder) I have copied the Lab71 folder to Lab72. We start by modifying the python script, in the "iterace_access_points()" function
  - Original section from Lab 71
  - Channel history panel
    Client count: `radio_stat → band_5 → num_clients`
    Current channel: `radio_stat → band_5 → channel`
    Channel width: `radio_stat → band_5 → bandwidth`
    TxPower: `radio_stat → band_5 → power`
  - AP info panel
    AP Name: `already saved in the $APName variable in Grafana`
    AP Model: `model`
    IP Address: `ip`
    External IP: `ext_ip`
    Mount direction: `mount`
    Power source: `power_src`
    Ethernet MAC: `mac`
    Radio MAC 5GHz: `radio_stat → band_5 → mac`
  - (Yes there are a lot of possibilities to optimize this code ☺)
  - Run the script and continue to next slide

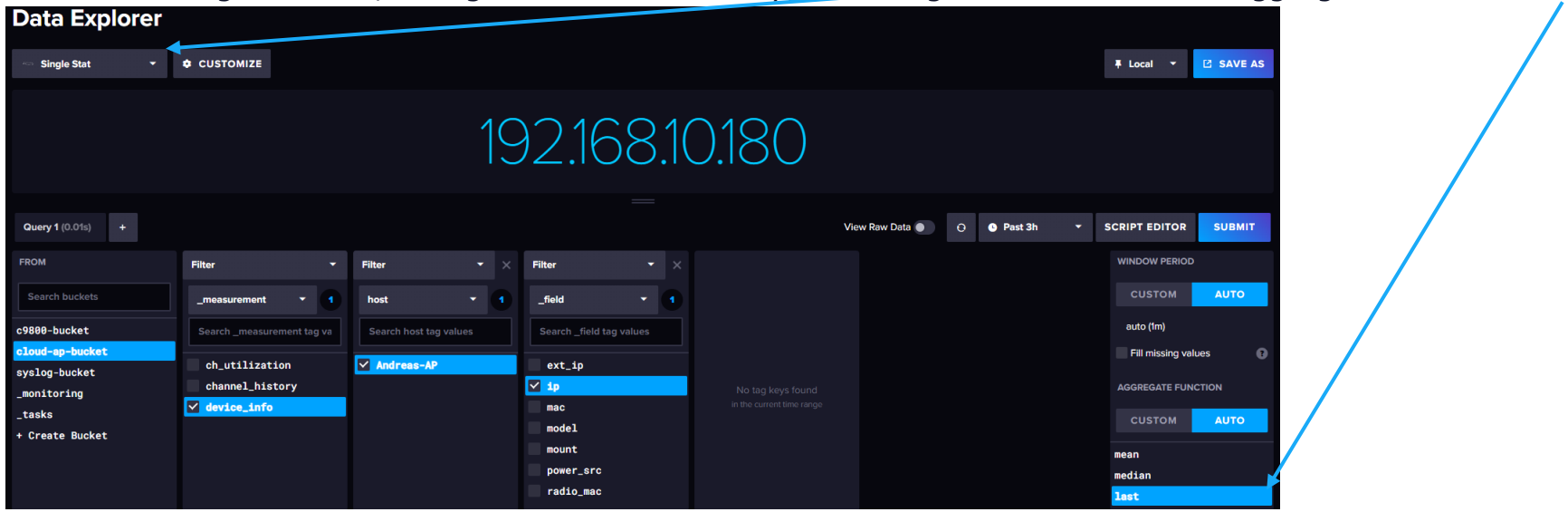
```
42  def iterate_access_points():
43      devices = get_device_statistics()
44      for device in devices:
45          if device['type'] == "ap":
46              interference = {}
47              for band in device['radio_stat']:
                    if band=="band_24": bandname = "2.4"
49                  if band=="band_5": bandname = "5"
50                  if band=="band_6": bandname = "6"
51                  interference[bandname+"GHz Wi-Fi"] = device['radio_stat'][band]['util_rx_in_bss'] + d
52                  interference[bandname+"GHz non-Wi-Fi"] = device['radio_stat'][band]['util_non_wifi']
53                  interference[bandname+"GHz Total"] = device['radio_stat'][band]['util_all']
54              update_influxDB('ch_utilization', device['name'], interference)
55
56              channel_history = {}
57              for band in device['radio_stat']:
58                  if band=="band_24": bandname = "2.4"
59                  if band=="band_5": bandname = "5"
60                  if band=="band_6": bandname = "6"
61                  channel_history[bandname+"GHz #Clients"] = device['radio_stat'][band]['num_clients']
62                  channel_history[bandname+"GHz Channel"] = device['radio_stat'][band]['channel']
63                  channel_history[bandname+"GHz Bandwidth"] = device['radio_stat'][band]['bandwidth']
64                  channel_history[bandname+"GHz TxPower"] = device['radio_stat'][band]['power']
65              update_influxDB('channel_history', device['name'], channel_history)
66
67              device_info = {}
68              device_info['model'] = device['model']
69              device_info['ip'] = device['ip']
70              device_info['ext_ip'] = device['ext_ip']
71              device_info['mount'] = device['mount']
72              device_info['power_src'] = device['power_src']
73              device_info['mac'] = device['mac']
74              device_info['radio_mac'] = device['radio_stat']['band_5']['mac']
75              update_influxDB('device_info', device['name'], device_info)
```


Checking the new data in InfluxDB

- Go to Data Explorer in InfluxDB and check that the new data is inserted there

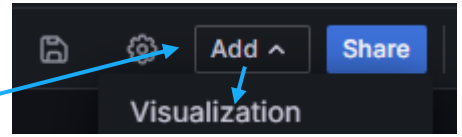


- Before clicking "Submit", change the visualization type to "single stat" and choose aggregation method "last"

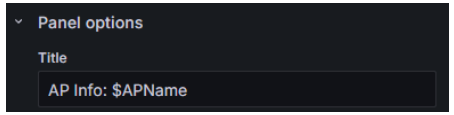


Adding the new values to our Dashboard in Grafana

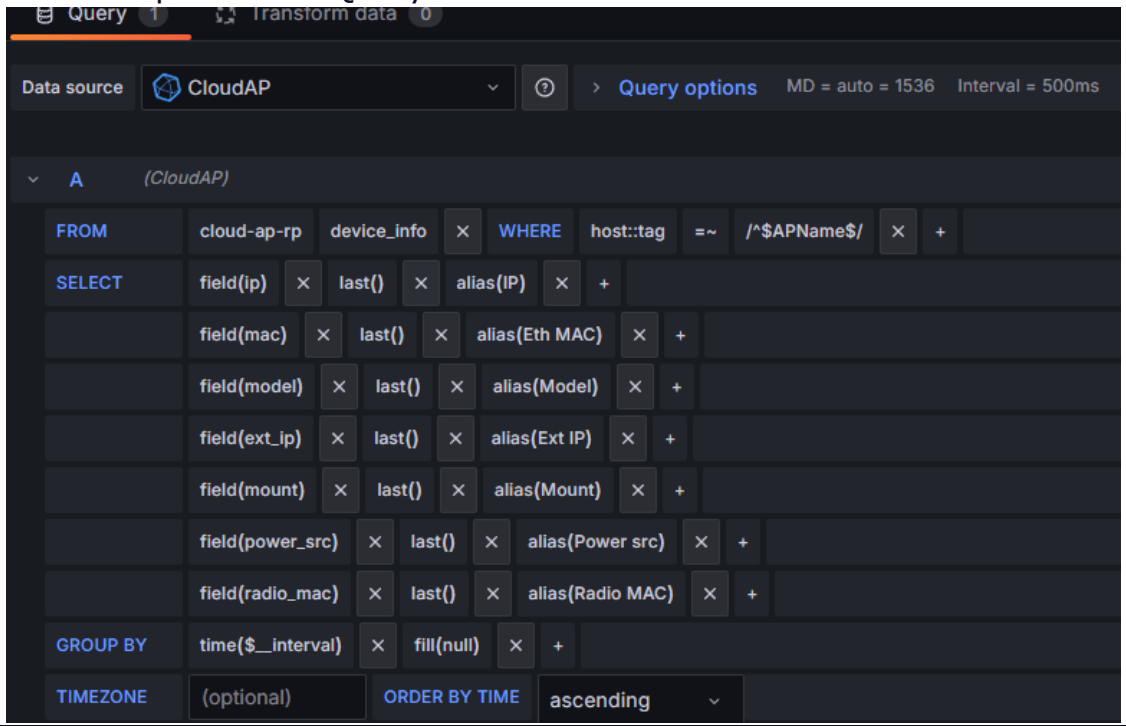
- In Grafana, go to Dashboards > Cound AP example dashboard
- We will first add a "AP info panel" on the top. Choose Add > Visualization



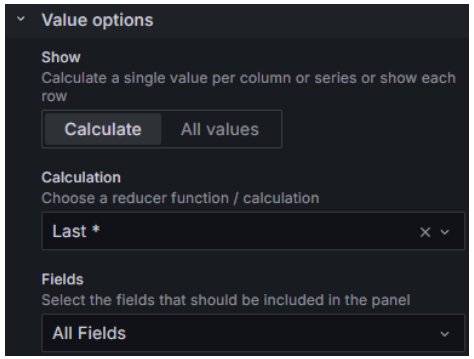
- Change "Panel Title" to "AP info: $APName"



- Populate the Query like this



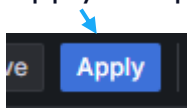
- Also, set the "Value options" to the right like this



- The live panel above the query should start populating data

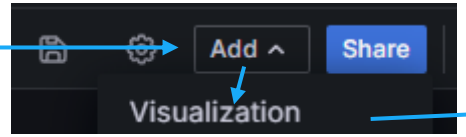
Adding the new values to our Dashboard in Grafana

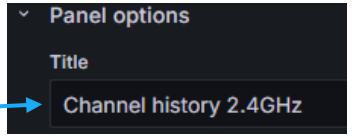
- Apply the panel, and resize it to cover the top of your Dashboard

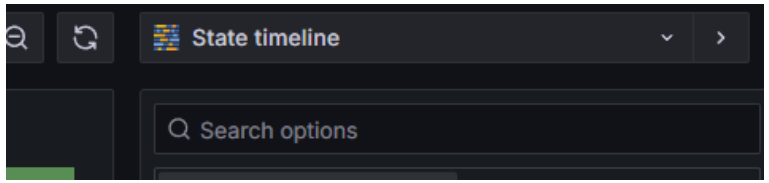


Adding the new values to our Dashboard in Grafana

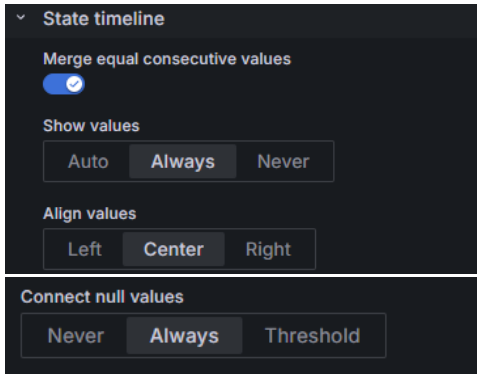
- Now create another Visualization
- Change "Panel Title" to "Channel history 2.4GHz"
- Change to "State timeline"



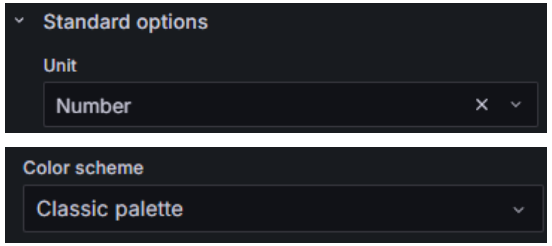




- Set State timeline options

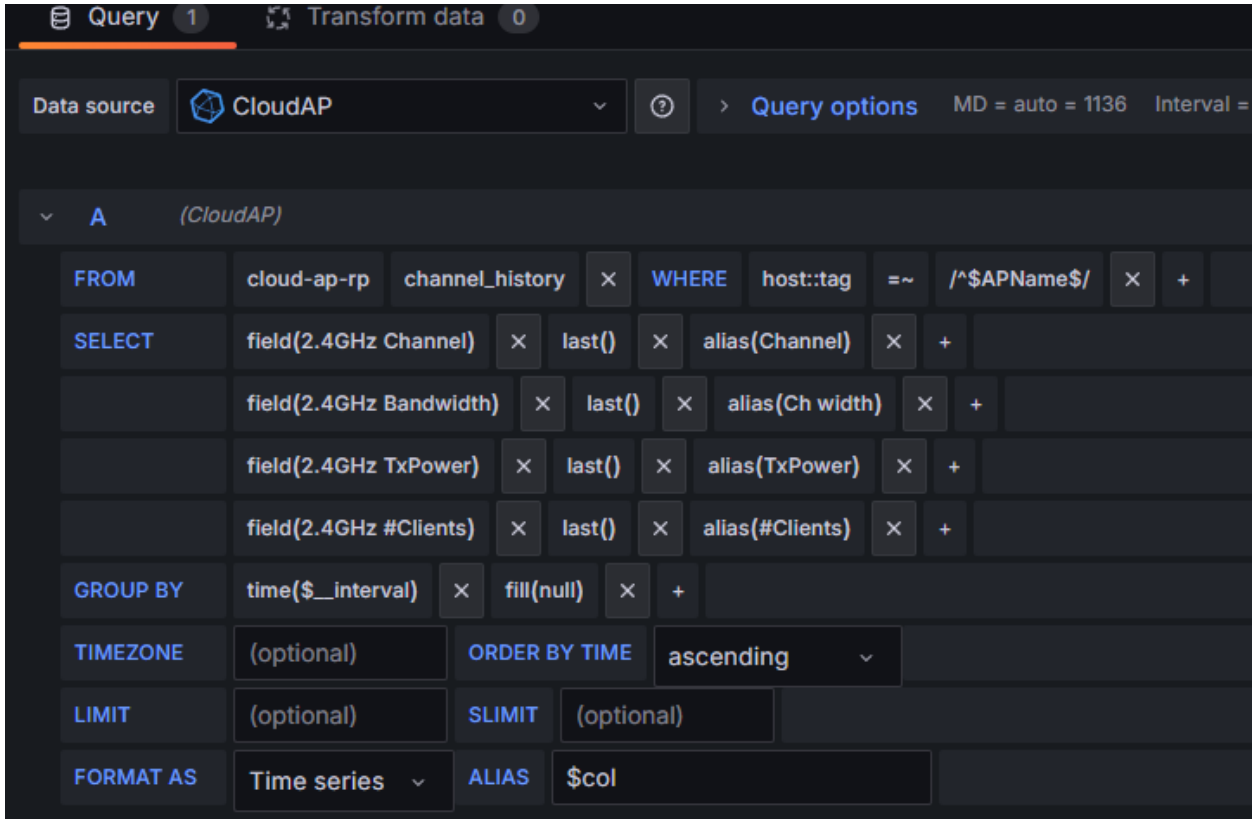


- Set Standard options



Adding the new values to our Dashboard in Grafana

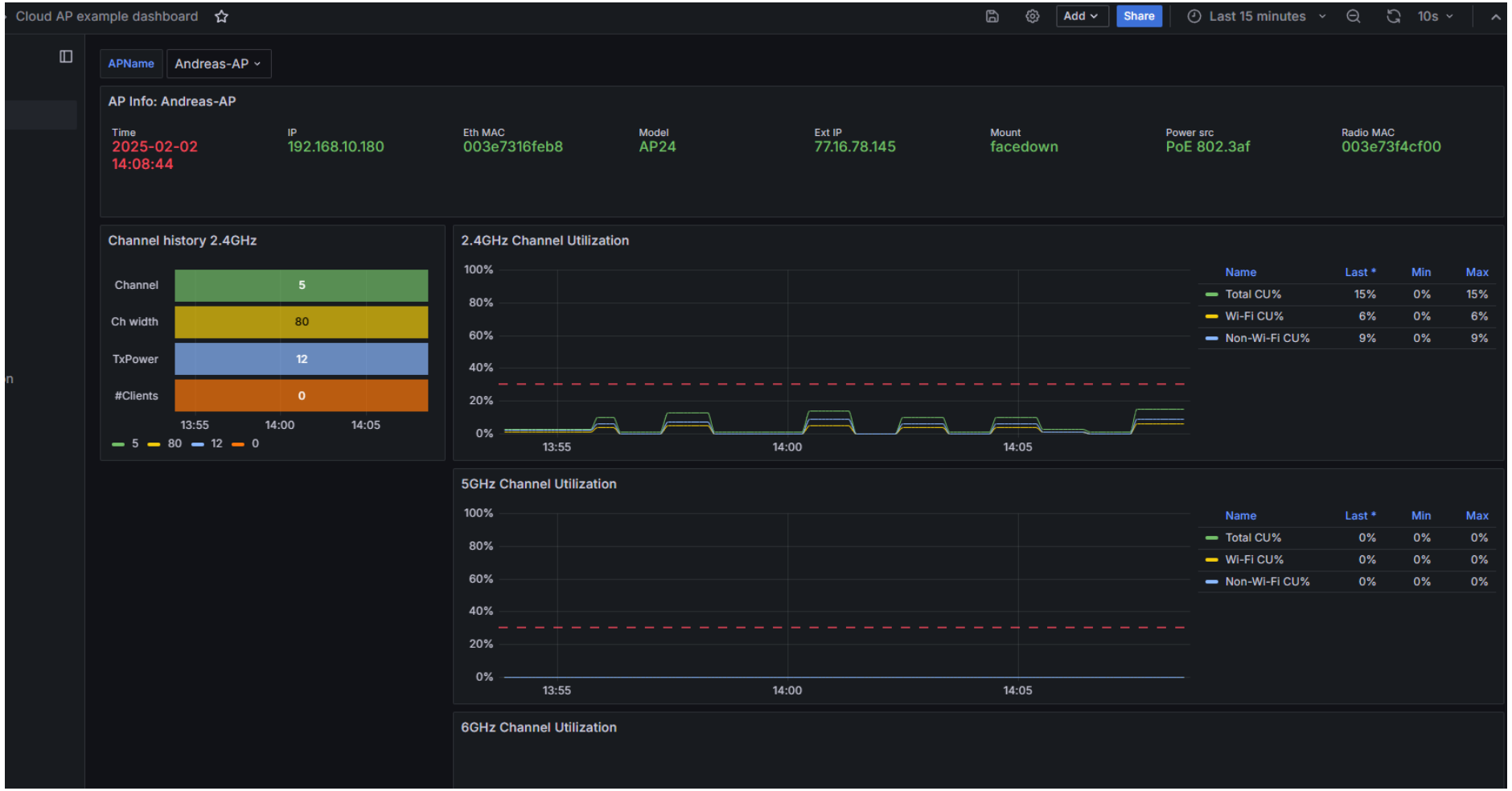
- Populate the Query like this. The live panel above should start populating data



- Apply to get back to your Dashboard

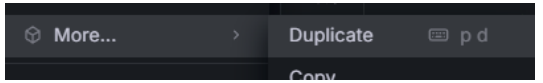
Adding the new values to our Dashboard in Grafana

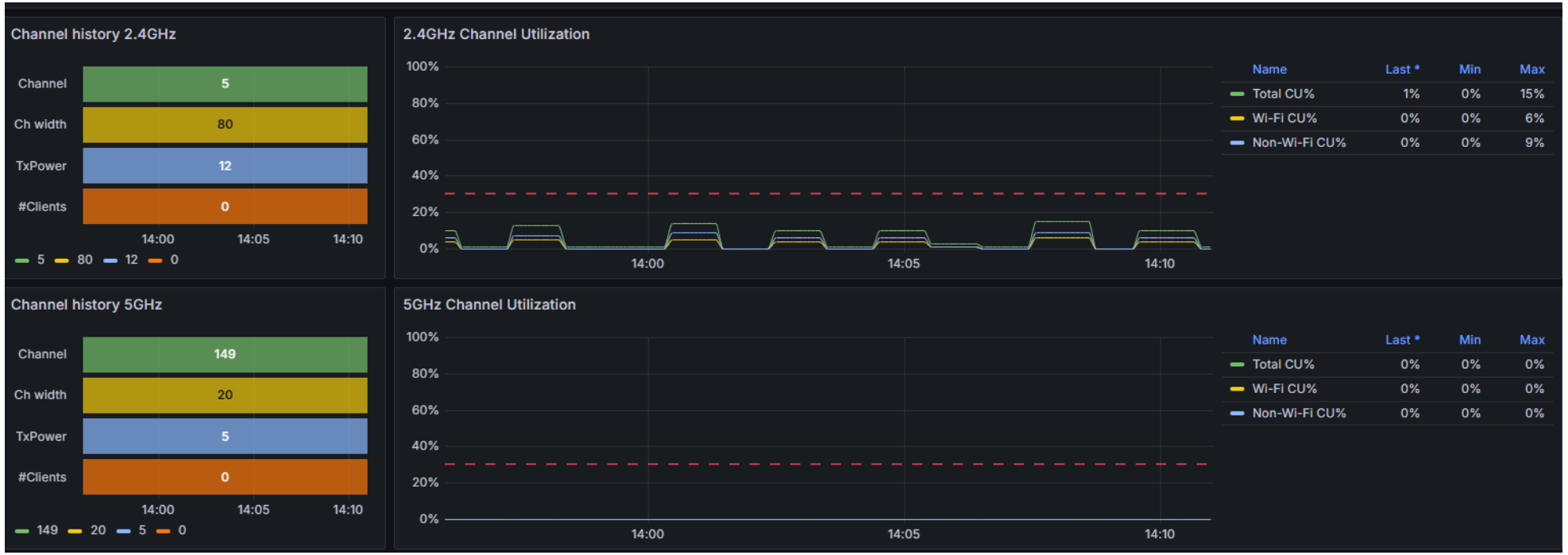
- Reorder and resize your panels to move your new panel to the left of the corresponding CU graphs



Adding the new values to our Dashboard in Grafana

- Duplicate the Channel history panel to create similar panels for 5 and 6GHz





- If you are very observant, you will notice that the "2.4GHz" stuff is actually showing 6GHz. It is because my MIST AP is a dual radio device, where one of the radios are 5GHz and the other is 2.4/6GHz. But this is just to show you some ideas, there are always a lot of things that can and should be tuned/changed for a production environment ☺