APIs By Example

Deep Dive, WLPC 2019 Phoenix.

albanom@google.com
Feb, 2019

Hope you didn't wait until now to do this:
$ docker pull mike909/wlpc-grpcserver:v1
$ docker pull mike909/wlpc-grpcclient:v1
$ docker pull mike909/wlpc-gnmiclient:v1
$ docker pull mike909/wlpc-gnmitarget:v1
$ docker pull mike909/influxdb:v1
$ docker pull mike909/grafana:v1

Files for this hosted on GitHub (https://github.com/mike-albano/wlpc-ocapi) including this deck.

OPENCONFIG

What's covered/Objective

Covered:

- YANG Modelling, Protocol Buffers, gRPC
- gRPC Network Management Interface (gNMI)
- Streaming Telemetry (Pub/Sub)
- TSDB Visualizations & Alerting

Objective:

- Introduction to the above.
- Go from zero to fully functioning systems, through lessons which build on each other.
- These are not production-ready systems, simply examples of usage and concepts.

OPENCONFIG

Prerequisites (Please tell me you did this already)

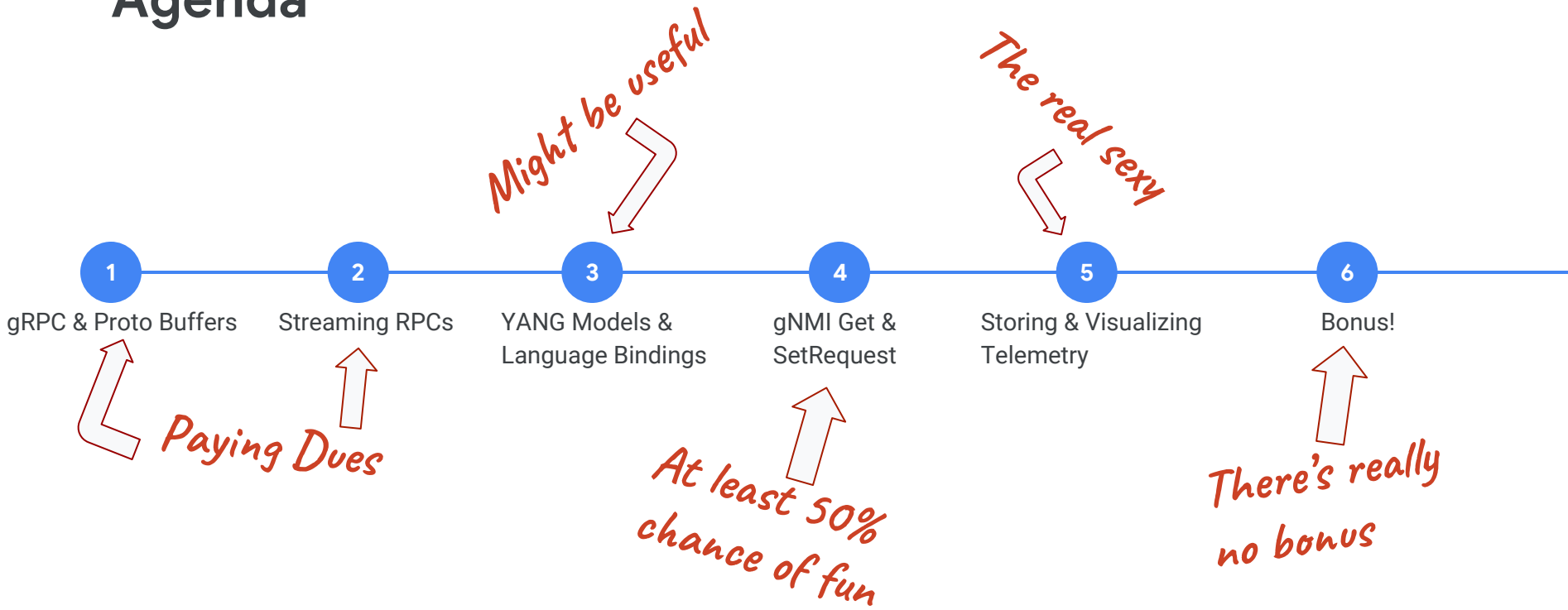
Does this work?

```
host_machine$ docker run hello-world
*snip*
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

(might need sudo if on Linux)

OpenConfig

Agenda



OPENCONFIG

Docker Tips

(Full cheat Sheet, last page of lesson plan)

Leave a container running, and attach a second console (in a new terminal window) by doing:

```
host_machine$ docker exec -it <container_id> /bin/bash
```

You obtain the container_id from the ‘docker ps’ command on your host. For example:

```
host_machine$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
3162670c9d94        grpc_client         "bash"              3 seconds ago
```

OPENCONFIG

Lesson 1.
gRPC & Protocol Buffers

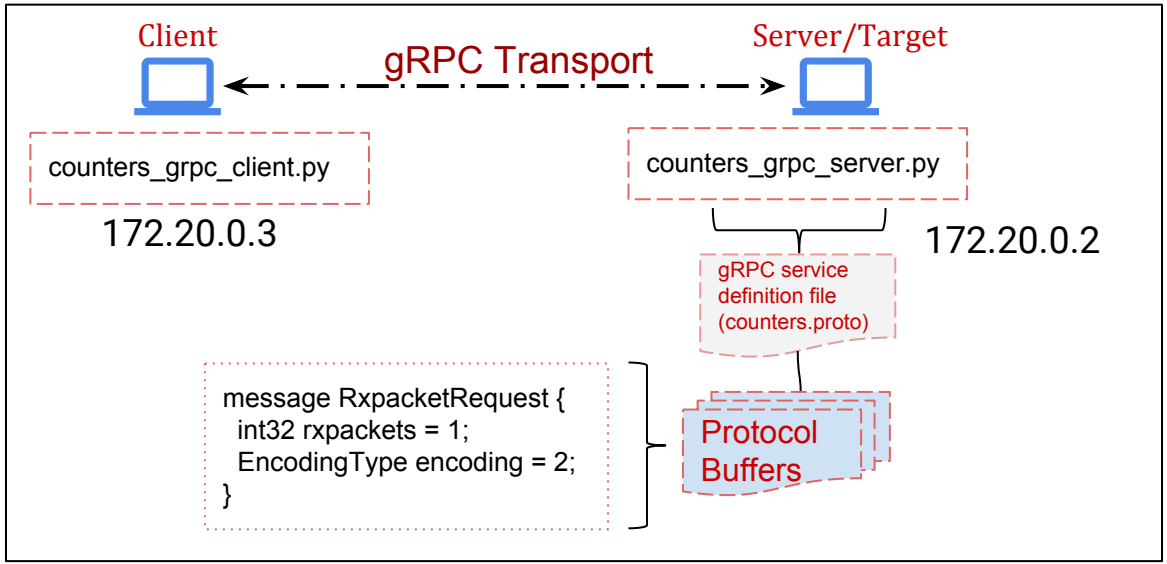
OPENCONFIG

Expected End Result

```
root@grpc_client:/home/user# python counters_grpc_client.py
message: "Rx packets for eth0: 26"
```

Note, your Rx packets number will differ.

Topology



OPENCONFIG

Set Up

1. Set up our virtualized network. This allows us to utilize static IPs.
   In a terminal in your host machine:

```
docker network create --subnet=172.20.0.0/24 ocnet
```

2. Run the Server container.
   In a terminal on your host machine:

```
docker run -it --net ocnet -h grpc_server --ip 172.20.0.2 mike909/wlpc-grpcserver:v1 /bin/bash
```

3. Run the Client container.
   In a **different terminal** on your host machine:

```
docker run -it --net ocnet -h grpc_client --ip 172.20.0.3 mike909/wlpc-grpcclient:v1 /bin/bash
```

   Linux: you might need to prepend with ‘sudo’.

OPENCONFIG

Lesson 1 Exercise 1
Write the protocol buffer.

This is a **language and platform neutral** definition for structured data, which in our case is a packet counter taken from the output of ‘ifconfig’.

There are two interfaces on the Target, Loopback and eth0. Define a message of type ‘String’ which we will used to determine which interface we run ‘ifconfig’ on.

For example:

```
message RxpacketRequest {
  string interface = 1;
}
```

counters.proto already in your Client container

OPENCONFIG

Lesson 1 Exercise 1

Write the protocol buffer.

Next, define a message for the Rx packets taken from **ifconfig**. Since this is a counter, represented as a positive number, this will be of type '**uint64**'. A full description of types can be found here.

For example:

```
message RxpacketReply {
  uint64 rxpackets = 1;
  string message = 2;
}
```

Why the "string message = 2"? This will be explained later in the lesson.

OPENCONFIG

Lesson 1 Exercise 1

Write the protocol buffer.

The full .proto file can be found in your **/home/user** directory in the **Client container**. You should have the Client container open in a terminal window, so you can peek at the file with:

```
root@grpc_client:/home/user# cat counters.proto
```

There is more metadata in the counters.proto file. For a full understanding of all these fields, see here.

OPENCONFIG

Lesson 1 Exercise 2

Compile source code from our counters.proto.

With our .proto now defined, we run the protocol buffer compiler.

We'll be using the Python language, though remember proto's are language (and platform) independent, so you could swap portions of these lessons with any of the supported languages.

The protobuff compiler has been installed for you within the Client and Server containers. Generate Python Classes which will be utilized by our Client & Server using the 'protoc' compiler:

```
root@grpc_client:/home/user# python -m grpc_tools.protoc -I./ --python_out=.
--grpc_python_out=. ./counters.proto
```

I skip the next 5 slides to save time (we instead focus on usage examples).

TLDR; we build language bindings from our .proto to use in our scripts.

OPENCONFIG

Lesson 1 Exercise 3

Write our client application

A small script which imports the generated protobuff code.

The focus here is on simplicity, not the Python code, however it may help to walk through the more important bits. If you prefer to skip ahead, simply peak at **/home/user/counters_grpc_client.py**.

```
channel = grpc.insecure_channel('172.20.0.2:50051')
```

As the variable name implies, this establishes an insecure gRPC channel to our Server (172.20.0.2) on port 50051.

OPENCONFIG

Lesson 1 Exercise 3

Write our client application

```
stub = counters_pb2.int_counterStub(channel)
```

Here we utilize the imported protobuf code, counters_pb2, which references the service int_counter (**see line 5 of counters.proto**) and creates a 'Stub'.

This stub is just a local object in Python.

Refer back to the counters.proto file for reference.

OPENCONFIG

Lesson 1 Exercise 3

Write our client application

```
response = stub.GetCounter(counters_pb2.RxpacketRequest(interface=server_int))
```

Here we utilize the ‘GetCounter’ RPC, **from the counters.proto**, and specify the interface we’re interested in getting Rx packets for. Note, this interface is passed into the ‘GetRxPackets’ function. For example, if you wanted to get Rx packets for the Loopback interface, you would simply change line 24 of “counters_grpc_client.py” to “lo”.

For example:

```
response = GetRxPackets('eth0') →  response = GetRxPackets('lo')
```

If you run this, for example “root@grpc_client# python counters_grpc_client.py”, **it will fail**.
That’s expected, since we have not set up the Server yet. Let’s do that!

OPENCONFIG

Lesson 1 Exercise 4

Write our server application

Over in your other terminal window (where you started the grpc_server container), the counters_pb2.py & counters_pb2_grpc.py files already exist. What we'll be doing is essentially the reverse of the Client app, which means we'll import the protobuffer code here as well.
We'll also need to write a small function to run 'ifconfig' and parse the output to get what we're interested in, which in this case is Rx packets on an interface.

Again, if you'd prefer to skip the explanation of the code, feel free to peek at **counters_grpc_server.py**. I'll highlight the important bits in the following slides.

OPENCONFIG

Lesson 1 Exercise 4
Write our server application

Take a look at **counters_grpc_server.py** in your wlpc-grpcserver container:

```
input = os.popen('ifconfig ' + request.interface)
```

counters_grpc_server.py

We're assigning the output of the entire ifconfig command to a variable named 'input'. The request message from the client written in Step 3 includes the interface we're interested in.

Remember the "string interface" in the proto?

This enumerates to "**ifconfig eth0**".

OPENCONFIG

Lesson 1 Exercise 4

Write our server application

```
rxcounter = int(''.join([x.split()[2] for x in input if 'RX packets' in x]))
```

We loop over the ifconfig output and assign the integer to our variable, **rxcounter**.

For example, ‘rxcounter’ would equal “31” in the following output:

```
root@grpc_server:/home/user# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.20.0.2  netmask 255.255.255.0  broadcast 172.20.0.255
        ether 02:42:ac:14:00:02  txqueuelen 0  (Ethernet)
        RX packets 31  bytes 3099 (3.0 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 14  bytes 1554 (1.5 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Lesson 1 Exercise 4
Write our server application

```
16      return counters_pb2.RxpacketReply(
17              message='Rx packets for %s: %i' % (request.interface, rxcounter))
```

Now we see why we assigned that string in the **counters.proto** file earlier. (string message = 2)

When we respond to the Client app over the RPC, we return two fields:

1. A string of our choosing (“Rx packets for…”)
2. An integer (the rxcounter).

Note: Responding with this concatenation of a string and an integer here is more for example, and won’t show up in later lessons when we get into YANG models and dealing with real Network Elements.

Lesson 1 Exercise 5
Put It Together

With some understanding of what the Client and Server are doing, let's run the Server by executing "python counters_grpc_server.py" on the "grpc_server" container. For example:

```
root@grpc_server:/home/user# python counters_grpc_server.py
gRPC server listening on port 50051
```

Let's run the Client by executing "python counters_grpc_client.py" on the "grpc_client" container. For example:

```
root@grpc_client:/home/user# python counters_grpc_client.py
```

What do you see?

OPENCONFIG

Lesson 1 Exercise 5
Put It Together

```
"Rx packets for eth0: 32"
```

That's it!

You are now getting "Telemetry" over a gRPC connection between a Client and a Server (I mean Target).

In further lessons you'll learn about YANG modelling & gNMI, but this is what's occurring at the lower level.

Extra credit; change the Rx packet value to a string, and see what happens.

Why does it fail? (see next slide)

Leave these containers running for Lesson 2.

OPENCONFIG

Lesson 1 Exercise 5
Extra Credit Answer

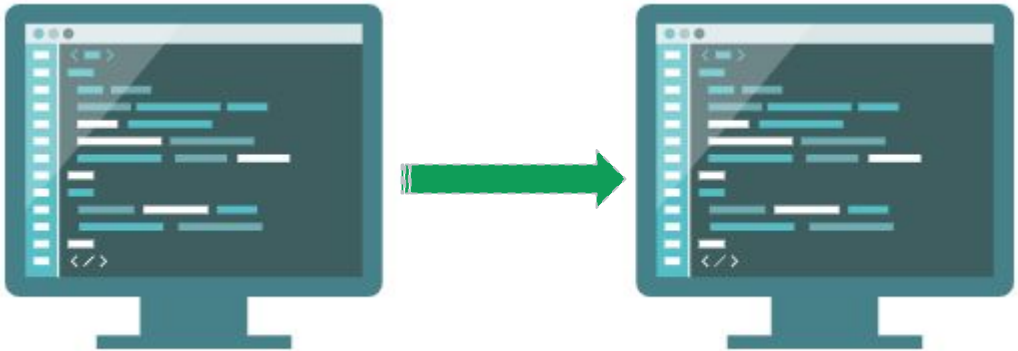
Answer

It *should* fail, as the proto strictly specifies the type as a ‘uint64’. Therefore, if you change the server app (counters_grpc_server.py) to assign the Rx packets as a string, you’d see something like this on the Client.

Later, you’ll see similar Schematic errors when we utilize YANG models and gNMI.

OPENCONFIG

Lesson 2.
Streaming RPC's



OpenConfig

Expected End Result

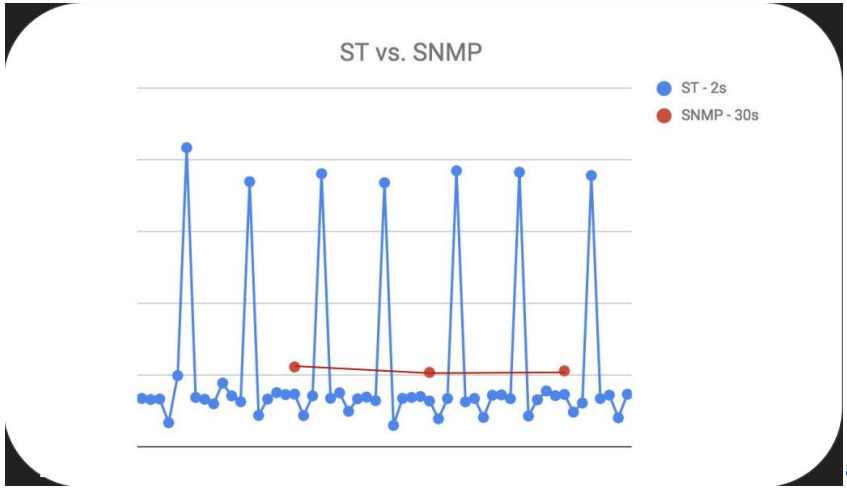
```
root@grpc_client:/home/user# python counters_grpc_streamclient.py
message: "Rx packets for eth0: 26"
message: "Rx packets for eth0: 27"
message: "Rx packets for eth0: 28"
...
```

Note, your Rx packets number will differ.

Lesson 2
Why Stream?

Streaming Telemetry, where a Publisher/Subscriber mechanism is used, is a big part of OpenConfig.

It changes the construct from a constant ‘Polling’ of Network Elements to one where a collection system (eg NMS) simply Subscribes to messages, reducing system overhead while increasing the granularity of Telemetry.



Source

Next up: a very basic example of this.

OPENCONFIG

Lesson 2 Exercise 1
Change The Proto

A slight modification to the proto file is all that's needed. We simply add the word "stream" to the rpc definition **in the proto file**. For example:

```
rpc GetCounter(RxpacketRequest) returns (stream RxpacketReply) {}
```

Note the slight difference between the two .proto files:
root@grpc_client:/home/user/**counters.proto** & **counters-stream.proto**

OPENCONFIG

Lesson 2 Exercise 1

Change The Proto

Don't forget to generate new source code from this proto file with:

```
root@grpc_client$ python -m grpc_tools.protoc -I./ --python_out=.
--grpc_python_out=. ./counters-stream.proto
```

This has already been done for you.

OPENCONFIG

Lesson 2 Exercise 2
Minor change to Client & Server

We slightly modifying the Client & Target app, to use new proto source, and introduce a loop.

The Python code is not the focus here, so you can simply look at "**/home/user/counters_grpc_streamclient.py**".

```
*snip*
  for r in response:
    print(r)
*snip*
```

This is all that's changed.

Similarly we added a while loop to "**root@grpc_server:/home/user/counters_grpc_streamserver.py**".

OPENCONFIG

Lesson 2 Exercise 2

Run The Server, Then The Client

On the grpc_server container:

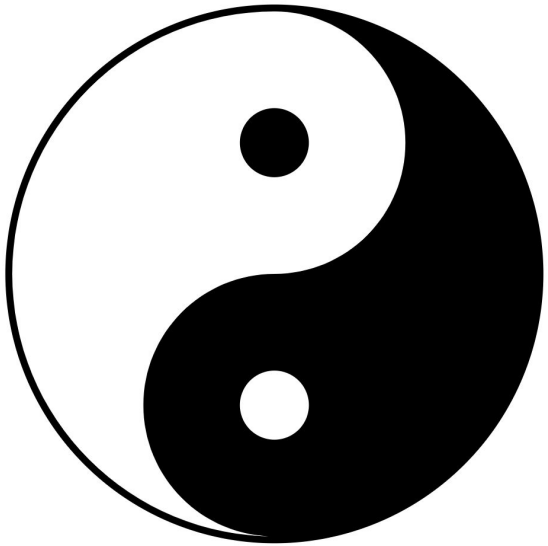
```
root@grpc_server:/home/user# python counters_grpc_streamserver.py
gRPC server listening on port 50051
```

And over on the grpc_client container:

```
root@grpc_client:/home/user# python counters_grpc_streamclient.py
"Rx packets for eth0: 54"
“Rx packets for eth0: 55"
"Rx packets for eth0: 58"
...
```

You’ve just used a Streaming RPC! You can now say things like this: I do Streaming Telemetry, I’ve used a Publisher/Subscriber (Pub/Sub) model, I built a Telemetry Collector. I can’t believe you still use SNMP. Wow.

Lesson 3.
YANG Models & Language Bindings



OpenConfig

Expected End Result

```
root@gnmi_client:/home/user# python interfaces_generator.py
{
  "openconfig-interfaces:config": {
    "enabled": true
  }
}
```

OPENCONFIG

Lesson 3 Exercise 1
Set Up

You can exit the containers used in the previous lesson with ctrl-D or “exit”.

Run the gNMI Client for use in this lesson, in a terminal on your host machine:

```
docker run -it --net ocnet -h gnmi_client --ip 172.20.0.3 --add-host
www.example.com:172.20.0.4 mike909/wlpc-gnmiclient:v1 /bin/bash
```

Note, **this is one line**.

OPENCONFIG

Lesson 3 YANG
Why?

We don't need to worry about writing protos or configuring gRPC when utilizing OpenConfig and gNMI.
Q: So, why did we just go through that?
A: Because gNMI is really just a proto describing gRPC messages for Network devices (APs).

- Next we'll be writing a YANG model to define the format of the Telemetry, which again in our case is Rx packets of an interface.

- YANG models as somewhat analogous to proto's; they define how data is supposed to look, giving it structure. YANG is a data modelling language, and like proto's, they are language-neutral.

- **Network Engineers** (that's us) write these OpenConfig YANG models in a vendor-neutral way.

OPENCONFIG

Lesson 3 Exercise 1

Write Our First YANG Model (Trouble sleeping? rfc7950, rfc6020)

Similar to the previous lesson, this won't be an all inclusive "how to write YANG models", just a taste.

Models start with meta, such as YANG version, namespace, prefix, etc. The interesting bit is the 'packet-counter' container. In YANG a container is an interior node in the Schema; which is to say, it doesn't have a value, like a "leaf" node does.
In the following, "packet-counter" is the container and "rx-packets" is the leaf node.

```
container packet-counter {
   description
     "Container to hold packet counters.";
   leaf rx-packets {
     type uint32;
     description
       "Received packets on the interface.";
   }
}
```

OPENCONFIG

Again, notice the 'type' here as an unsigned 32bit integer.

Lesson 3 Exercise 2

Make our YANG Model adhere to OpenConfig Style

The full YANG model is viewable at “**/home/user/interfaces.yang**”.

This small model is enough to generate language bindings from, and use in our code, however we’re going to make this model adhere to the OpenConfig Style first...

This is really about structure of the YANG containers, groups, leaf nodes etc. We’re also going to add a leaf node for “enabled”. Specifically this is:

```
leaf enabled {
  type boolean;
  description
    "This leaf enables and disabled the interface.";
}
```

Lesson 3 Exercise 2

Make our YANG Model adhere to OpenConfig Style

Take a look at "**/home/user/openconfig-interface.yang**".

- Config and State are differentiated within groupings and containers, but are within the same model. An operator (that's us) may want to obtain both the **State of the configuration** on the device, and the **'actual' State** of the device. These can be distinctly different (eg. just because you told a device to disable an interface does not necessarily mean it did so).

- Every Config leaf has a corresponding State leaf, however the reverse is not true. In our example, rx-counters is a State leaf only. (You wouldn't configure the received packets on an interface).

- The series of groupings and containers allows for maximum re-use. You'll see examples of this throughout OpenConfig models, where one Model may import another model. Eg, our interfaces model here imports openconfig-extensions (also present in your **/home/user** directory).

OPENCONFIG

Lesson 3 Exercise 3

Verify Our YANG Model Does Not Contain Errors.

This is commonly referred to as 'linting'. For this we utilize pyang (with the OpenConfig plugin). This is already on your client system, so give it a shot:

```
root@gnmi_client:/home/user# pyang --plugindir
./oc-pyang/openconfig_pyang/plugins/openconfig.py openconfig-interface-broken.yang
--strict --lint -p ./

oc-interfaces.yang:26: error: unterminated statement definition for keyword
"description", looking at r
```

There's an error in the model. Show the fix!

OPENCONFIG

Lesson 3 Exercise 3

Pyang

Other handy options in pyang.

View the Model as an ASCII tree on your terminal by adding the flag “**-f tree**” to the previous command.

The output would be:

```
module: openconfig-interface
  +--rw interfaces
     +--rw config
     |  +--rw enabled?   boolean
     +--ro state
        +--ro enabled?      boolean
        +--ro rx-packets?   uint32
```

Here it becomes more obvious what we meant before by “every config leaf has a state leaf, but not vice-versa”. ‘rx-packets’ is ONLY within the state container, while ‘enabled’ is present in both.

OPENCONFIG

Lesson 3 Exercise 4
Generate Language Bindings

Now that we have our YANG model, let's try to use it.

- YANG is language-neutral, so generate your bindings in language you're writing your Client with.
  PyangBind for Python
  YGOT for Go.

- This should feel similar to what we saw earlier with proto's.
  Instead of generating source code from a proto (using protoc compiler), we're generating source code from a YANG model (using PyangBind).

OPENCONFIG

Lesson 3 Exercise 4
Generate Language Bindings

Since PyangBind is already installed for you, simple run the following command to generate our Python source.

```
root@gnmi_client:/home/user# pyang --plugindir $PYBINDPLUGIN -f pybind -o
interface_bindings.py openconfig-interface.yang
```

Paraphrasing from the README:

- $PYBINDPLUGIN is the location of the PyangBind plugin.
- interface_bindings.py is the desired output file.
- openconfig-interface.yang is the path to the YANG module that bindings are to be generated for.

Lesson 3 Exercise 5
Use The Language Bindings

We now have “/home/user/**interface_bindings.py**”. If you poke around in there, you’ll recognize some of the classes and function names, but it’s not important to understand this file. We’ll just be importing it into our Client.

Let’s write a small script which imports our language bindings (**interface_bindings.py**) and prints some JSON to our terminal. The full script is in **/home/user/interface_generator.py**.

Lets go over the interesting bits...

OPENCONFIG

Lesson 3 Exercise 5
Use The Language Bindings

```
configs = openconfig_interface()
def CreateConfigs():
  int_conf = configs.interfaces
  int_conf.config.enabled = True
```

Here we are assigning the imported bindings to a variable named ‘configs’.

We use that inside the CreateConfigs function by assigning the ‘interfaces’ Class to the variable “int_conf” (recall the “**container interfaces**” in the YANG model we wrote earlier).

We can now manipulate this configs object, by setting config leaf’s. In this example we’re setting the interface **enabled** boolean to “**True**”. This would result in the Target (Network Element) enabling the interface.

OPENCONFIG

Lesson 3 Exercise 5
Use The Language Bindings

```
print(pybindJSON.dumps(all_configs, indent=2))
```

Here we are using PyangBinds function to ‘serialize’ the configs object as JSON, and print that to stdout.

This is an oversimplified contrived example; in the reality the YANG model would be much more verbose, containing lists [of interfaces], types, etc., as would the Client software and resulting JSON payload, but the **core concepts are the same**.

Lesson 3 Exercise 5
Use The Language Bindings

Run the script. You should see the following:

```
root@gnmi_client:/home/user# python interface_generator.py
{
  "config": {
    "enabled": true
  }
}
```

OPENCONFIG

Lesson 3
Conclusion

- What we did:

  YANG model ⇒ OpenConfig YANG model ⇒ generating language bindings ⇒ using those bindings in a small script, which prints our OpenConfig conforming JSON.

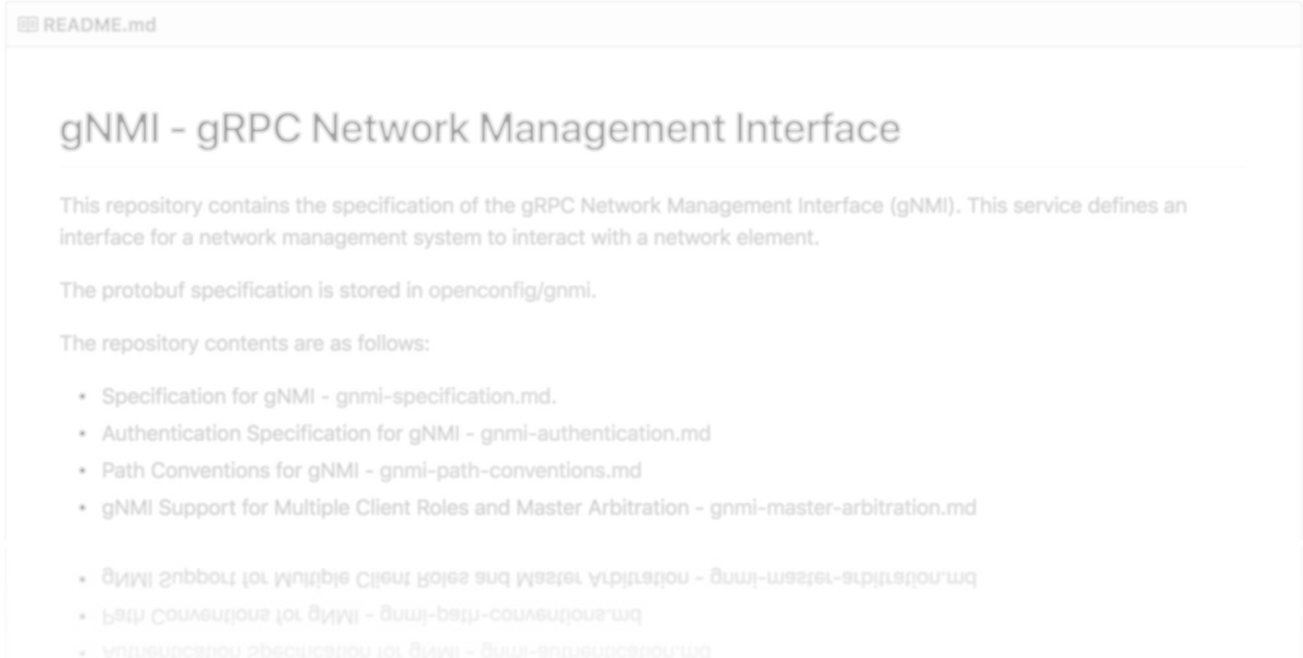
  The JSON is exactly what would end up getting sent to the gNMI Target (eg the AP, Router, Switch) to enable an interface.

- Similar to before, you can purposefully break this by utilizing the incorrect ‘type’ in the script. For example, “enabled” is of type ‘boolean’. **What happens if you change that to a string?**
- Later, we do the reverse. We will deserialize, or rather, take JSON and load it into a class object (called ‘configs’ in our example) while **ensuring it adheres** to the YANG model.

  Leave the gnmi_client container running for the next lesson.

OPENCONFIG

Lesson 4.
gNMI. Get and SetRequest.



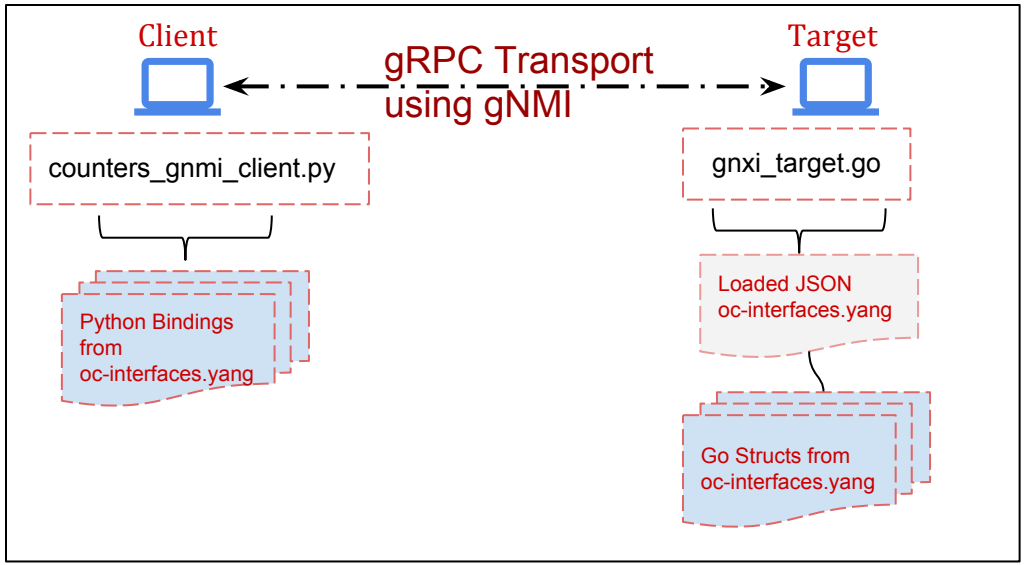
OPENCONFIG

Expected End Result

```
root@gnmi_client:/home/user# python simple_client.py
notification {
  timestamp: 1537939439031070110
  update {
    path {
      elem {
        name: "interfaces"
      }
      elem {
        name: "state"
      }
      elem {
        name: "rx-packets"
      }
    }
    val {
      uint_val: 73
    }
  }
}
```

OPENCONFIG

Topology



OPENCONFIG

Lesson 4

Set Up

In your host machine, run:

```
docker run -it --net ocnet -h gnxi_target --ip 172.20.0.4 mike909/wlpc-gnmitarget:v1
```

Note, we left off the '/bin/bash' so you should see something similar to the following in your terminal:

```
1 gnmi_target.go:119] starting to listen on :10161
1 gnmi_target.go:125] starting to serve
```

Our emulated Target has loaded a JSON payload in memory, which includes a key/value for the state leaf for "rx-packets". Ordinarily this is an AP. Leave it running for the duration of this lesson.

You **already** have the gnmi_client container from the last lesson.

If you closed it, you can restart the container with:

```
docker run -it --net ocnet -h gnmi_client --ip 172.20.0.3 --add-host www.example.com:172.20.0.4
mike909/wlpc-gnmiclient:v1 /bin/bash
```

OPENCONFIG

Lesson 4 Exercise 1

Create the gNMI Source Files

Just like we did back in Lesson 1, we're going to generate the source code by running the 'protoc' compiler. This time however, we'll point to the gNMI proto, instead of our own proto. The gNMI repo, with necessary proto files is already in your **/home/user** directory, so you can simply run the following from /home/user **on your gnmi_client container**:

```
root@gnmi_client:/home/user# python -m grpc.tools.protoc
--proto_path=/home/user/gnmi/proto/gnmi_ext/
--proto_path=/home/user/gnmi/proto/gnmi/ --python_out=./client_app
--grpc_python_out=./client_app /home/user/gnmi/proto/gnmi/gnmi.proto
```

You should now see additional files in /home/user/client_app (gnmi_pb2.py & gnmi_pb2_grpc.py). Just like in Lesson 1, these are the source-code files which we've generated from the .proto. These files will be imported into our Client app.

OPENCONFIG

--Time Check-- If you're approaching 2 hours, consider skipping the next 4 slides.

Poll the audience -- who cares how RPC Stubs & channels work Vs. just wants to run stuff and see examples?

Lesson 4 Exercise 2

Set up the Client & Issue GetRequests

As before, you can peek at the final code for this portion of the exercise in **/home/user/simple_client.py**.
We will go over the interesting bits here:

```
import gnmi_pb2
import gnmi_pb2_grpc
import gprc
```

Firstly, we are importing our protoc-generated code from the previous step.

```
channel = grpc.secure_channel('www.example.com:10161', creds)
```

This should look very **familiar** to Lesson 1 - Exercise 3.

The added creds variable in use here is due to our sample gNMI Target's usage certificates.
We defined creds on the previous line.

OPENCONFIG

Lesson 4 Exercise 2

Set up the Client & Issue GetRequests

We then define the gRPC Stub and move on to formulate the gNMI GetRequest (in a function called … GetRequest):

```
stub = gnmi_pb2_grpc.gNMIStub(channel)
```

Again, this should look very familiar as we did the same thing in Lesson 1.

Lesson 4 Exercise 2

Set up the Client & Issue GetRequests

Now we'll begin to formulate some gNMI calls.

```
gnmi_response = GetRequest(stub, 'interfaces')
```

Here we're assigning a variable to what we will ultimately 'Get' from the gNMI Target (the other container running gnxi_target).
"interfaces" here represents the xpath, which should look familiar, as it's the first YANG container -- Line 55 of openconfig-interface.yang.

Paths are a big part of gNMI, and their use will become obvious in the coming lesson.

Lesson 4 Exercise 2

Set up the Client & Issue GetRequests

In the GetRequest function we see the following gNMI specific helpers:

```
path_list = [gnmi_pb2.PathElem(name=path, key={})]
paths = gnmi_pb2.Path(elem=path_list)
response = stub.Get(gnmi_pb2.GetRequest(path=[paths]), metadata=[
    ('username', 'foo'), ('password', 'bar')])
```

- The first line utilizes the imported proto-generated code (gnmi_pb2) to formulate an array (Python List) of gNMI Path's we will supply in our gNMI GetRequest. The 'path' in 'name=path' is passed in to this function; ('interfaces').
- The second line is constructing our paths into Path Elements. See gnmi path conventions for more info here. Path encoding is critical to gNMI; this will become more clear in the next exercises.
- The third line is initiating the GetRequest. This includes some required metadata, like username/password which our gNMI Target is expecting for authentication.
- "key={}" is used when we have a leaf-list in our YANG model, in order to reference an instance within the list. Eg "key={eth1}". See here for more info.

OPENCONFIG

Lesson 4 Exercise 2

Set up the Client & Issue GetRequests

Give it a shot. Run the **simple_client.py**:

```
root@gnmi_client:/home/user# python simple_client.py
notification {
  timestamp: 1537927476628149440
  update {
    path {
      elem {
        name: "interfaces"
      }
    }
    val {
      json_val: "{\"config\":{\"enabled\":true},\"state\":{\"enabled\":true,\"rx-packets\":73}}"
    }
  }
}
```

You just initiated a gNMI GetRequest, received a GetRespone, and printed that response to the terminal (stdout). The above is the protobuffer response. You can probably recognize the ‘json_val’ content.

Next, we’ll turn this into JSON.

OPENCONFIG

Lesson 4 Exercise 3

Convert the protobuf response to JSON

To make the GetResponse a bit easier to read, let's turn it into JSON (and print it). We're also going to index into the protobuf response, since we're only after what's contained in the 'json_val' in the above protobuf response.

Add the following lines to the bottom of simple_client.py:

```
js_response = json.loads(gnmi_response.notification[0].update[0].val.json_val)
print(json.dumps(js_response, indent=2))
```

You should also comment out the previous print statement; "print(gnmi_response)". This will make the output contain **only** JSON.

Lesson 4 Exercise 3

Convert the protobuf response to JSON

Your output should now resemble this:

```
{
  "state": {
    "rx-packets": 73,
    "enabled": true
  },
  "config": {
    "enabled": true
  }
}
```

You should experiment with changing the ‘print(gnmi_response)’ line of code.

For example, how would you print out only the timestamp, or only the rx-packets of the GetResponse message?

OPENCONFIG

--Time Check-- If you're past 2 hours, consider skipping the next 3 slides.

Just tell everyone "paths" are a thing, and they're important -- you'll see why later.

OPENCONFIG

Lesson 4 Exercise 4

Utilize gNMI Paths

Let's modify **simple_client.py** to provide a gNMI Path in our GetRequest.

gNMI Paths are used to limit the GetRequest to something more granular than '/' (eg everything from the root of the OpenConfig Tree).

The appropriate way to do this would be through a helper function, but we'll just hard-code it here:

```
path_list = [gnmi_pb2.PathElem(name='interfaces', key={}),
        gnmi_pb2.PathElem(name='state', key={}),
        gnmi_pb2.PathElem(name='rx-packets', key={})]
```

Now, instead of using a single path ("interfaces"), we're supplying a list of gNMI PathElements.

OPENCONFIG

Lesson 4 Exercise 4

Utilize gNMI Paths

If you look at the response you'll now see the value of our GetResponse is limited to simply 'rx-packets'...or 73. If you've done it correctly, your output should resemble:

```
root@gnmi_client:/home/user# python simple_client.py
notification {
  timestamp: 1537939439031070110
  update {
    path {
      elem {
        name: "interfaces"
      }
      elem {
        name: "state"
      }
      elem {
        name: "rx-packets"
      }
    }
    val {
      uint_val: 73
*snip*
```

OPENCONFIG

Lesson 4 Exercise 4

Utilize gNMI Paths

If you did not comment out/delete the previous lines where you loaded the Response as JSON (js_response) the script will fail, as the current GetResponse is now returning a uint_val (not a json_val).

Just change line 41 to be “print(gnmi_response)” again.

How could you change this, if you **did** want to print only the current GetResponse value of 73?

OPENCONFIG

Lesson 4 Exercise 5

Deserialize the Response

We already have our Python language bindings from the previous lesson, but now we're going to utilize them to ensure the response we get from the gNMI Target is adhering to our YANG model.

--Time Check-- If you're past 2 hours skip "hands-on" in next 2 slides: Just talk through it.

A real-world example of this might be for an operator to **ensure** their previously configured network element, is still configured as expected.

This process of taking the response JSON and placing it back 'in to' our language bindings from Lesson 3 is referred to as 'deserializing' (See here).

To do this, we can basically **combine our two sample scripts** (simple_client.py & interface_generator.py). You can peak at the final version of this in **/home/user/notso_simple_client.py**.

OPENCONFIG

Lesson 4 Exercise 5

Deserialize the Response

Basically, the interesting bits are in these lines:

```
path_list = [gnmi_pb2.PathElem(name='interfaces', key={}),
             gnmi_pb2.PathElem(name='config', key={})]

js_response = json.loads(gnmi_response.notification[0].update[0].val.json_val)
pybindJSONDecoder.load_json(js_response, None, None, obj=configs.interfaces.config)
```

- The first two lines show we're doing a gNMI GetRequest while specifying the path of "/interfaces/config".
- We use PyangBinds pybindJSONDecoder to load the above JSON into our configs object. **Again, this enforces strict adherence** to the YANG model we created back in Lesson 3.

OPENCONFIG

Lesson 4 Exercise 5

Deserialize the Response

Similar to what we did in Lesson 3, you can print the contents of this configs object (which now contains the response from the gNMI Target) as JSON with:

```
  print(pybindJSON.dumps(configs, indent=2))
```

$ python notso_simple_client.py:

```
{
  "interfaces": {
    "config": {
      "enabled": true
    }
  }
}
```

OPENCONFIG

Lesson 4 Exercise 6

It's time to get real.

Ok, we're finally going to configure these things...

Open a terminal and run InfluxDB in it's own container:

```
docker run -it --net ocnet --ip 172.20.0.5 mike909/influxdb:v1
```

Leave that container open for informational messages.

OPENCONFIG

Lesson 4 Exercise 6

Provision your AP.

- This is a “day-0” operation.
- This is the ‘only’ time we care about the MAC address of an AP. Forever after this, we configure/monitor APs based on hostname (eg FQDN), not MAC address.
- No profiles, no “discovery” of ID’s prior to each SetRequest (common in RESTful). This is key!
    - Operators are in charge of “profiles” or logical “groupings” of configuration. RF Profiles, ap-groups, VAPs, SSID Profiles, AAA server-groups, etc. etc. No more!
    - gNMI/OC. API written **FOR** Network Elements.

Change the following lines in wlpc-gnmi.py:

```
210  gnmi_target = '<target_here>:8080'  # Put your APs IP and target-port here.
211  ap_name = 'AP01-albano.example.net'  # This is your APs FQDN? Make sure its unique!
212  ap_mac = '00:11:74:87:C0:7F'  # Your APs eth MAC (printed on AP)
```

OPENCONFIG

Lesson 4 Exercise 6

Provision your AP.

-- Time Check -- If past the 2.5 hour point, OR people are bored/hungry; just run wlpc-gnmi.py so we can do fun stuff (Telemetry/monitoring) on day 2

- Hey, these bits in configs_lib.py should look familiar!

```
provision_apconfigs = ap_manager_configs.provision_aps.provision_ap
day0 = provision_apconfigs.add(ap.mac)
    day0.config.mac = ap.mac
    day0.config.hostname = ap.ap_name
    day0.config.country_code = 'US'
    json_value = json.loads(pybindJSON.dumps(day0, mode='ietf', indent=2))
```

Provision the AP with:

```
gnmi_client$ python wlpc-gnmi.py --mode provision
```

Or do-it-yourself with the gnmicli.py client; see next slide.

OPENCONFIG

Lesson 4 Exercise 6

Provision your AP - DIY.

- Do it yourself by first generating the JSON:

```
$ python wlpc-gnmi.py --mode provision --dry_run
```

Then use the py_gnmicli.py utility to issue a SetRequest to your AP:

```
python py_gnmicli.py -t <your_ap> -p <ap_port> -x /provision-aps/provision-ap[mac=<your_mac>]/ -m
set-update -user admin -pass admin -o openconfig.mojonetworks.com -g -val @dry_run_provision.json
```

These common commands are in your container: cat sample_py_gnmicli_commands

If your using a Mist AP:

1. Remove the “-o” and “-g” options.
2. Your target “-t” will be: openconfig.mist.com, and your port “-p” will be 443.
3. Your username will be “admin@example.net” (password “admin”)

OPENCONFIG

Lesson 4 Exercise 7

Lay down the config!

--Time Check-- It's OK to start with this on Day 2, then roll right into Telemetry & Visualizations.

Automatically, with the following script:

```
python wlpc-gnmi.py --mode configure
```

Or Do It Yourself example:

```
python wlpc-gnmi.py --mode configure --dry_run
```

Followed by:

```
python py_gnmicli.py -t <your_ap|cloudthing> -p 8080 -m set-update -user admin -pass admin -x
/access-points/access-point[hostname=<your_ap_name>]/ -g -o openconfig.mojonetworks.com -val
@dry_run_configs.json
```

OPENCONFIG

Lesson 4 Exercise 7

Complete.

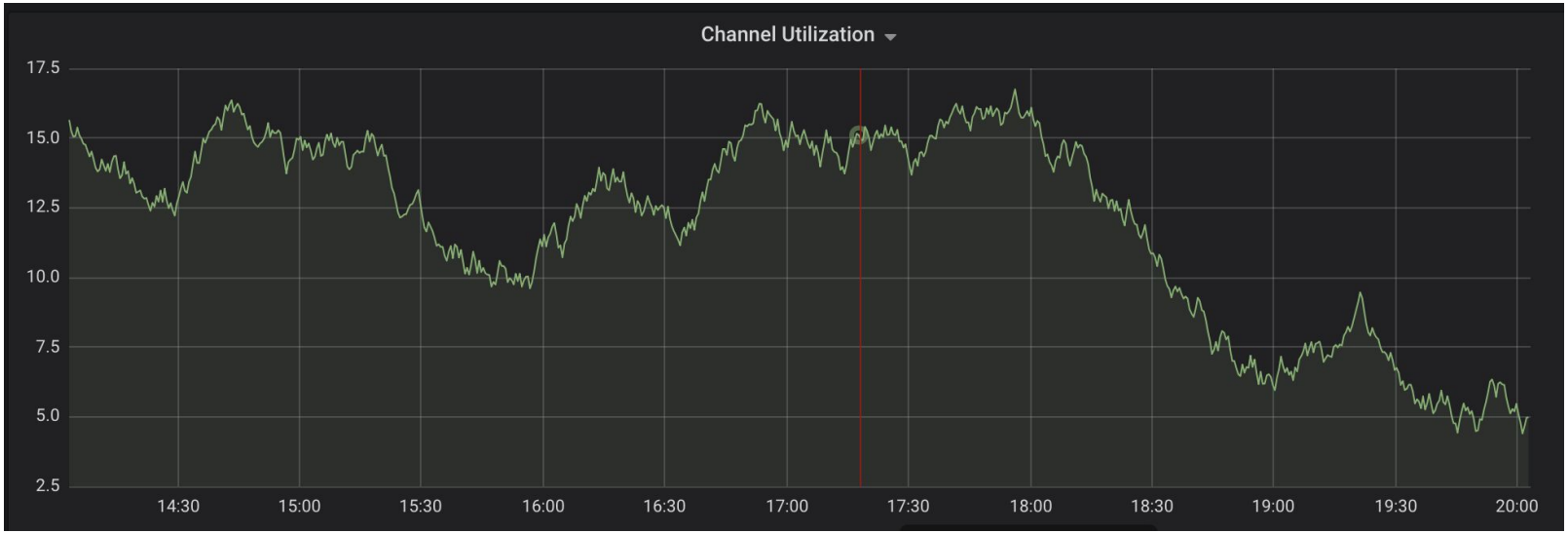
- **If you applied configuration yourself,** run “python wlpc-gnmi.py --mode configure” to populate the DB with desired configuration.
- Leave the gnmi_client container and InfluxDB container running for the next lesson.
- You can close the gnmi_target container.

OPENCONFIG

Lesson 5.
Storing & Visualizing Telemetry

OPENCONFIG

Expected End Result



OPENCONFIG

Lesson 5

Set Up

Your InfluxDB container **should still be running**. If not, run it again…

```
docker run -it --net ocnet --ip 172.20.0.5 mike909/influxdb:v1
```

You'll need to run wlpc-gnmi.py --mode **configure again** to populate the DB.

Open another terminal to run Grafana in it's own container:

```
docker run -dp 3000:3000 --net ocnet --ip 172.20.0.6 mike909/grafana:v1
```

Here we'll be showing visualizations using an Open Source TSDB, InfluxDB and Grafana.

OPENCONFIG

Lesson 5 Exercise 1

Populate the Database

- Add 802.11 CU% as a metric (channel_utilization) & write to DB.
- Add the config & state as a metric (config_state) & write to DB.
- Compare intended configuration Vs. actual configuration & write to DB.

We'll be using the InfluxDB Python client library to populate the DB.

```
def main(unused_argv):
227  dbclient = _create_db()  # Create DB and dbclient.
229  config_state = _get(ap, 'config_state')  # Get root of tree for AP.
230  dbjson = _prep_json(config_state, 'config_state', ap)  # Prep it for DB write.
231  _write_db(dbclient, 'ap_telemetry', dbjson)  # Write State JSON to DB.
232  _config_diff(dbclient, 'ap_telemetry', ap)  # Compare State Vs Intent.
```

Now we're ready to start populating data and graphing it...

OPENCONFIG

Lesson 5 Exercise 1

Continuously run Gets

Run “**wlpc-gnmi.py**” scrip, in your gnmiclient container:

```
python wlpc-gnmi.py --mode monitor
```

Your output should resemble:

```
I0109 20:55:57.500288 139712081696512 wlpc-gnmi.py:154] DB write successful
I0109 20:55:57.569354 139712081696512 wlpc-gnmi.py:198] config in sync
I0109 20:55:57.579210 139712081696512 wlpc-gnmi.py:154] DB write successful
I0109 20:55:57.974092 139712081696512 wlpc-gnmi.py:234] Channel Utilization: 0
...
```

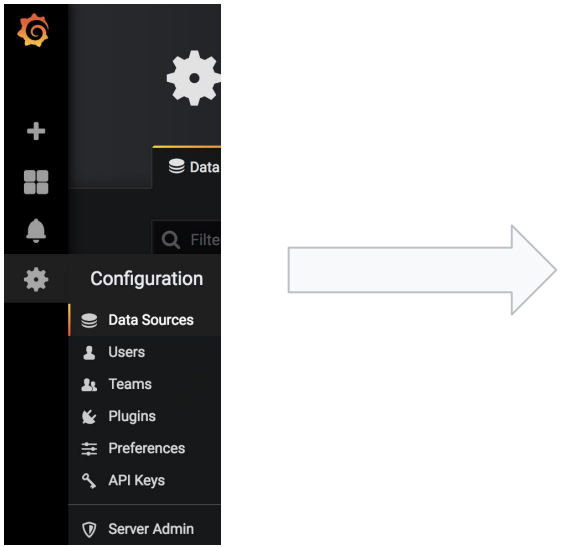
Lesson 5 Exercise 2

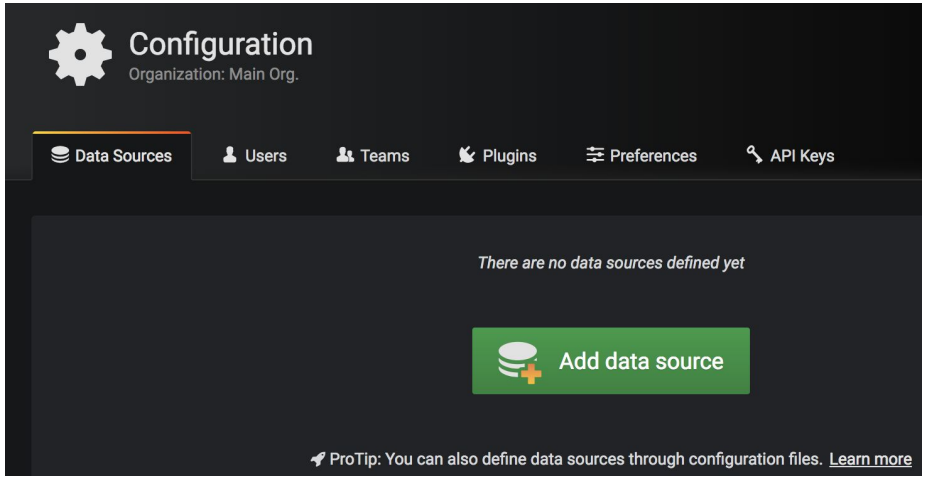
Visualizations

Set up Grafana, and point it to our TSDB data source.

Browse to http://localhost:3000. Default user/pass is admin/admin.

Click Configure-->Data Sources, Add New Data Source:





OPENCONFIG

Lesson 5 Exercise 2

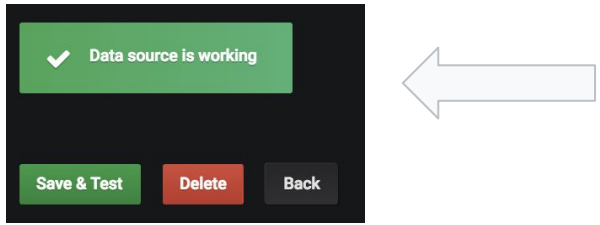
Add the TSDB data-source

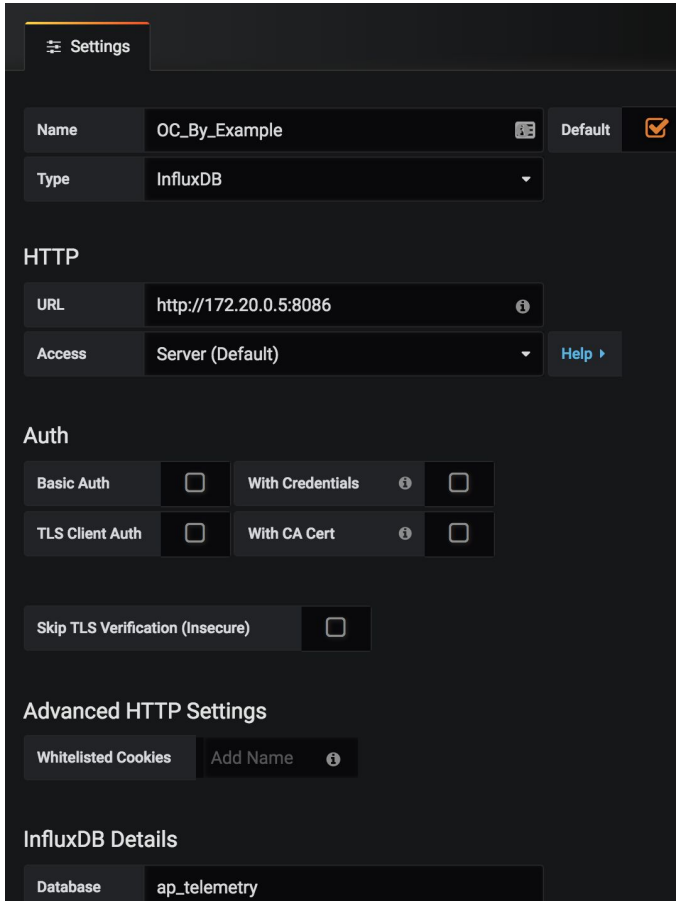
The options are mostly self explanatory.

Here are the highlights:

- Type: InfluxDB
- Name: OC_By_Example
- URL: http://172.20.0.5:8086
  (our Container running InfluxDB)
- Database: ap_telemetry

Click Save & Test, and you should see “Data source is working”:





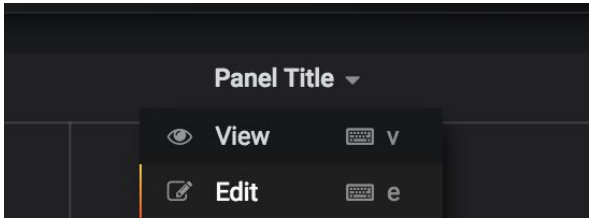
OPENCONFIG

Lesson 5 Exercise 3

Graph It!

Lets graph the Telemetry we are consuming, eg Channel Utilization.

1. Click Dashboards➔ Add New Dashboard. We'll be using a dashboard type of 'Graph'.
2. Click 'Panel Title'-->Edit:



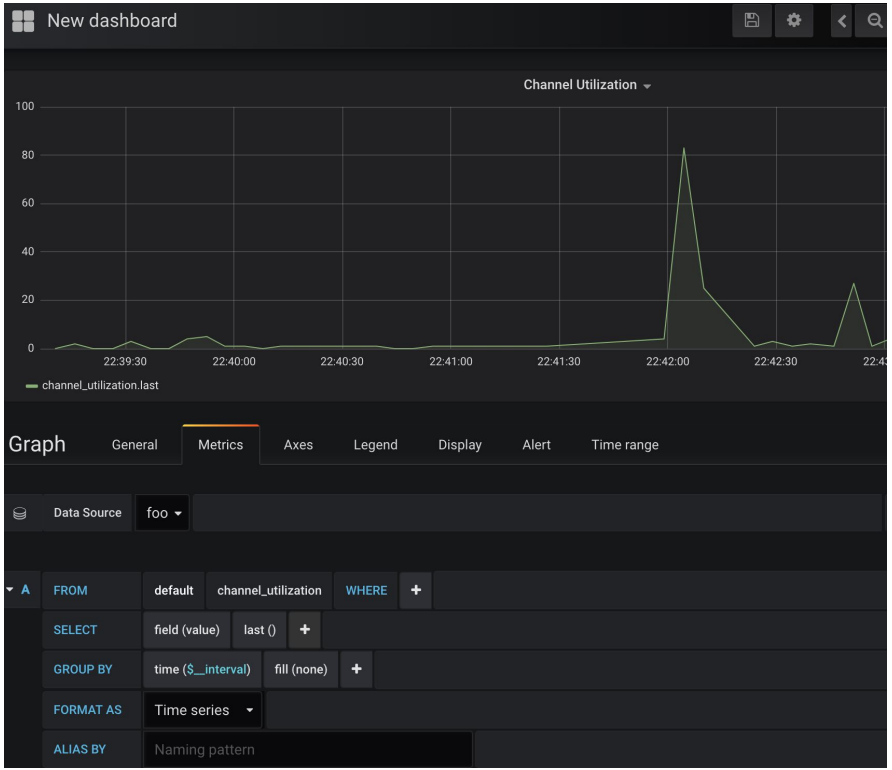
3. Change Data Source to what you named it in Step 1 (OC_By_Example)
4. Click in the 'Select Measurement' field. It should pre-populate with our metrics that are being inserted by our script, including 'channel_utilization'.
5. Change additional params as needed:
   fill(none)
   select(last)

OPENCONFIG

Lesson 5 Exercise 3

Graph It!

You should see the graph beginning to populate itself with the results.



OpenConfig

Lesson 5 Exercise 4

Create Alert

We're going to set up a channel utilization alert which will fire an update to a Slack channel.

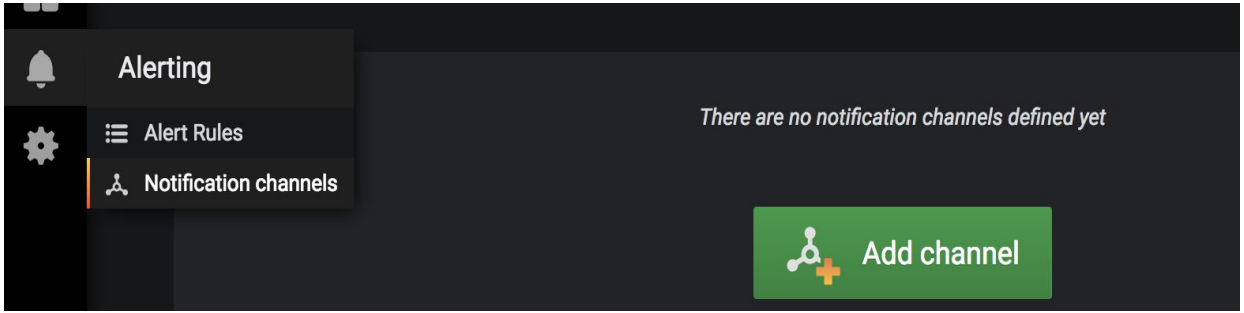
1. Go to Alerting ➔ Notification Channels ➔ Add channel.
2. Use the following settings:

   Type: Slack

   Url: https://hooks.slack.com/services/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

   Token:

   xoxp-498636724544-498770958785-500777105558-105daeba7851ff0f6c7eba7ff69ac600xoxp-498636724544-498770958785-498643190752-40bcfc957e0074e1a3124229698ffbd1



OPENCONFIG

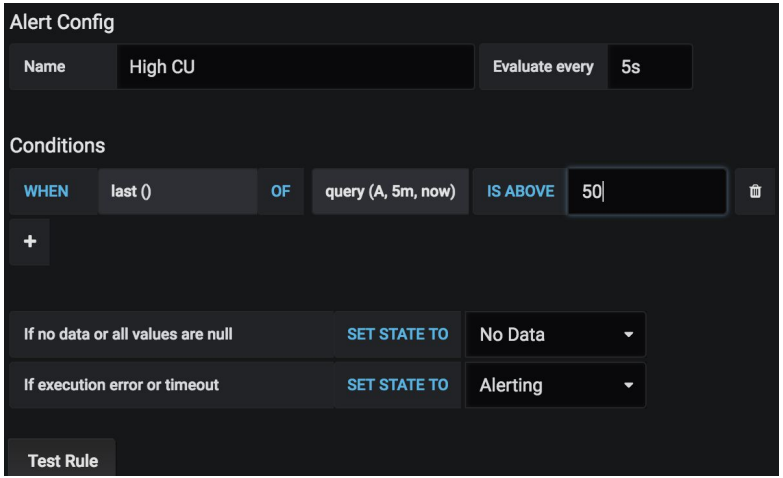
Click here if you'd like to join the Slack channel yourself.

Lesson 5 Exercise 4

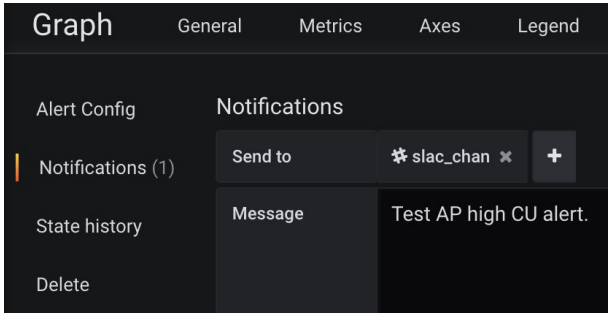
Create Alert Logic

Go back to your graph to set up the alert conditions (High Channel Utilization)

1. Name it
2. Evaluate every 5s
3. Change Conditions to last()
4. Set your threshold (IS ABOVE)
5. Add the Slack Notification



Optionally, change “Notification” message to Include your ap-name.



OPENCONFIG

Lesson 5 Exercise 4

Generate an Alert

Spike the CU and see your alert fire to the Slack channel.

iperf3 -c 192.168.1.2 -t60

OPENCONFIG

Lesson 5 Exercise

Repeat for Configuration Mismatch

Repeat the previous exercise on your own, except for a config_sync issue:

- Graph config mismatch
- Set up the Alert
- Generate an alert by changing a channel with py_gnmicli. For example:

```
python py_gnmicli.py -t openconfig.mist.com -p 443 -m set-update -x
/access-points/access-point[hostname=<your_apname>]/radios/radio[id=0]/config/chann
el -user admin@example.net -pass admin -val 36
```

Hint: cat sample_py_gnmicli_commands

Hint2: docker exec -it <container_id> /bin/bash

The database is already being populated for you (metric: config_sync).

OPENCONFIG

Closing

Obviously almost all the tools, concepts and descriptions here are introductory.

Hopefully this was enough to draw some conclusions about the advantage of vendor/language/tool-chain neutrality and freedom of choice in all aspects.

Appendix / Supplemental

Lesson 5 Exercise 1

Populate the Database

If your real Target (AP) is not working for some reason, you can use the mock gnmi_target to graph counters.

- Add our ‘rx-packets’ as a metric.

Over on your gnmi_client container, we’re going to use the InfluxDB Python client library.

Add a couple lines to our “**notso_simple_client.py**” script, within the main function. For example:

```
if __name__ == '__main__':
  client = InfluxDBClient('172.20.0.5', 8086, 'root', 'root', 'oc_by_example')
  client.create_database('oc_by_example')
```

Now we’re ready to start populating data…

OPENCONFIG

Lesson 5 Exercise 1

Populate the Database

Populate the DB with "**client.write_points(<data_points>)**". Add that into our script, and put it in a loop so we continually add the rx-packets value, every 5 seconds.

Replace our 'path_list' variable, and add the following in **notso_simple_client.py**:

```
path_list = [gnmi_pb2.PathElem(name='interfaces', key={}),
             gnmi_pb2.PathElem(name='state', key={}),
             gnmi_pb2.PathElem(name='rx-packets', key={})]
while True:
  gnmi_response = GetRequest(stub, path_list)
  rx_packets = gnmi_response.notification[0].update[0].val.uint_val
  print('Adding rx-packets %s to TSDB' % rx_packets)
  client.write_points([{"measurement": "rx_packets", "fields": { "value":
      rx_packets}}])
  time.sleep(5)
```

OPENCONFIG

Lesson 5 Exercise 1

Populate the Database

You should recognize most of this.

We've experimented in previous lessons with dealing with the gNMI GetResponse, changing paths etc.. The only new line here is the '**client.write_points...**', which as the name implies sends the result of our Gets to our TSDB. When you run this, it should show the following in your terminal:

```
root@gnmi_client:/home/user# python notso_simple_client.py
Adding rx-packets 73 to TSDB
Adding rx-packets 73 to TSDB
...
```

Leave that script running, as it will continually add data points to the TSDB using the InfluxDB HTTP API.