



Enterprise

Software Defined Networking

Keywords

Software Defined Networking, Network Programability, Python, Controller, Network Abstraction, OpenDayLight, NETCONF, RESTCONF, JSON, API, YANG Data Modeling, Model Driven.

References

- Cisco PRNE & NPDESI
<https://www.cisco.com>
- Python Docs
<https://www.python.org/doc/>
- OpenDayLight Docs
<http://docs.opendaylight.org>

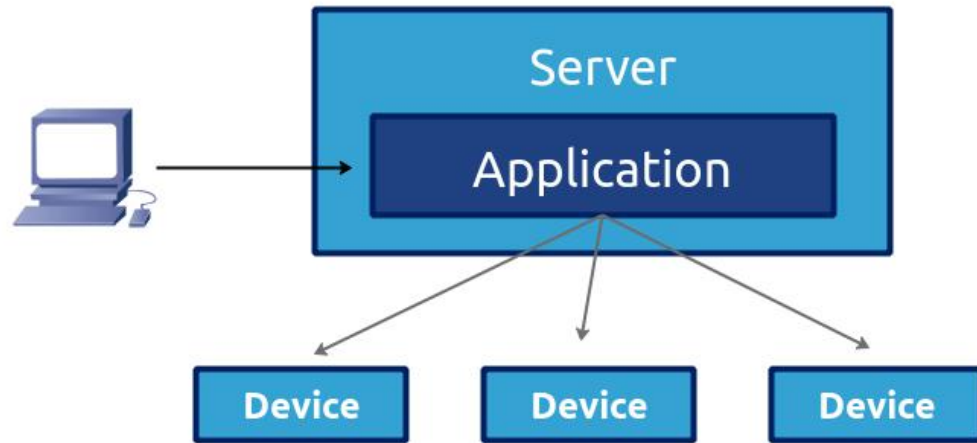


Network Programmability

Introduction

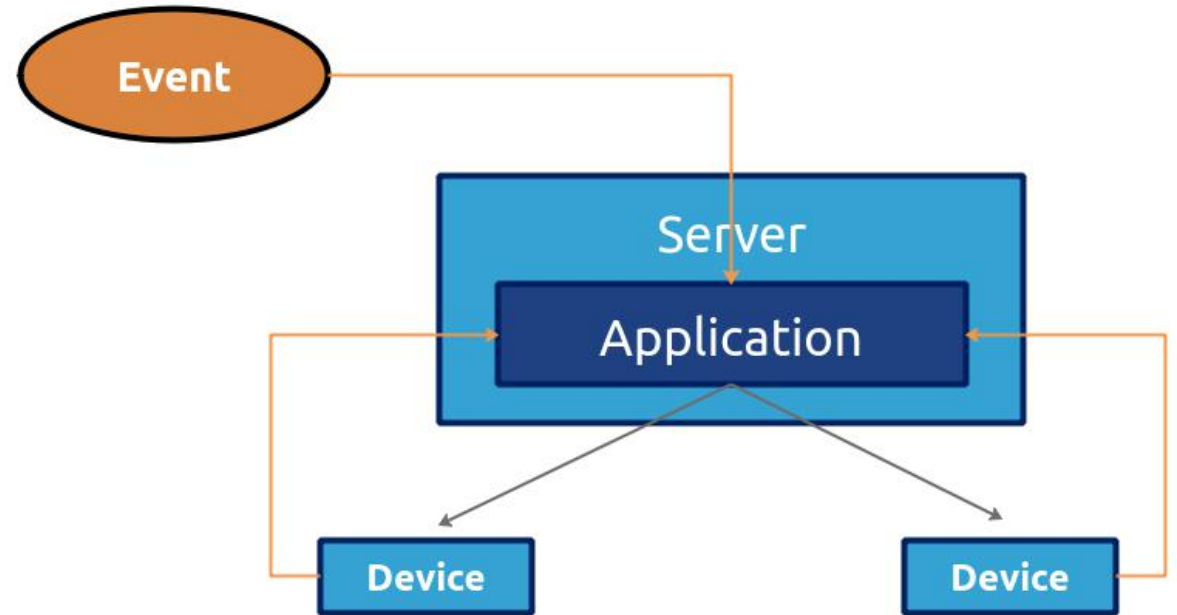
- Network Programmability is a set of tools or apps to deploy, manage, and troubleshoot a network device.
- A programmability-enabled network is driven by intelligent software that can deal with a single node or a group of nodes.
- The term network programmability can have different meanings, to a network engineer, programmability means interacting with a device to driving a configuration or do some troubleshooting.

Proactive vs Reactive



Proactive

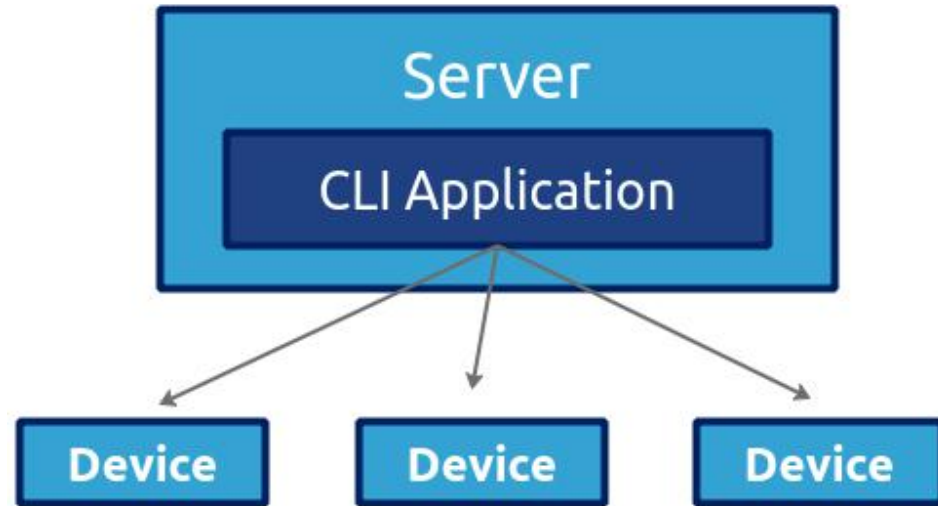
- Manual Change
- Change by human



Reactive

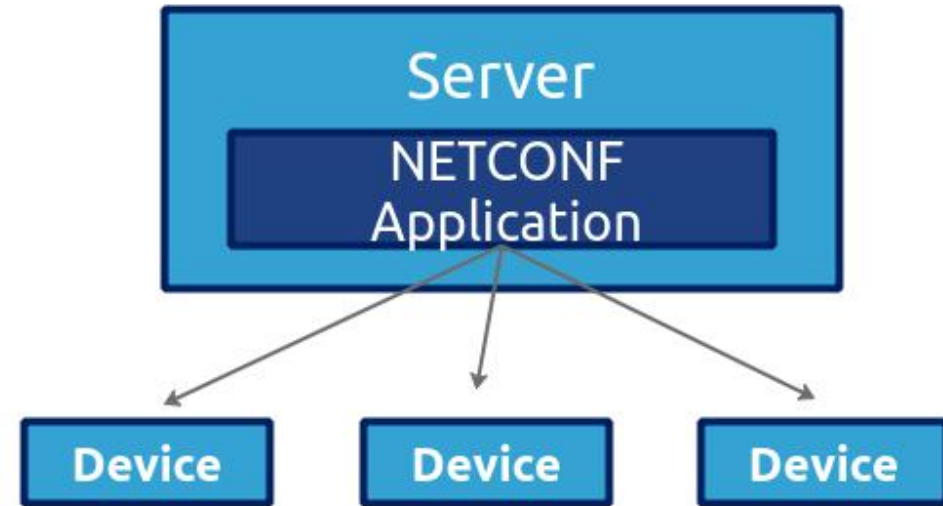
- Change by event
- Device send information to server

CLI vs Netconf



CLI Application

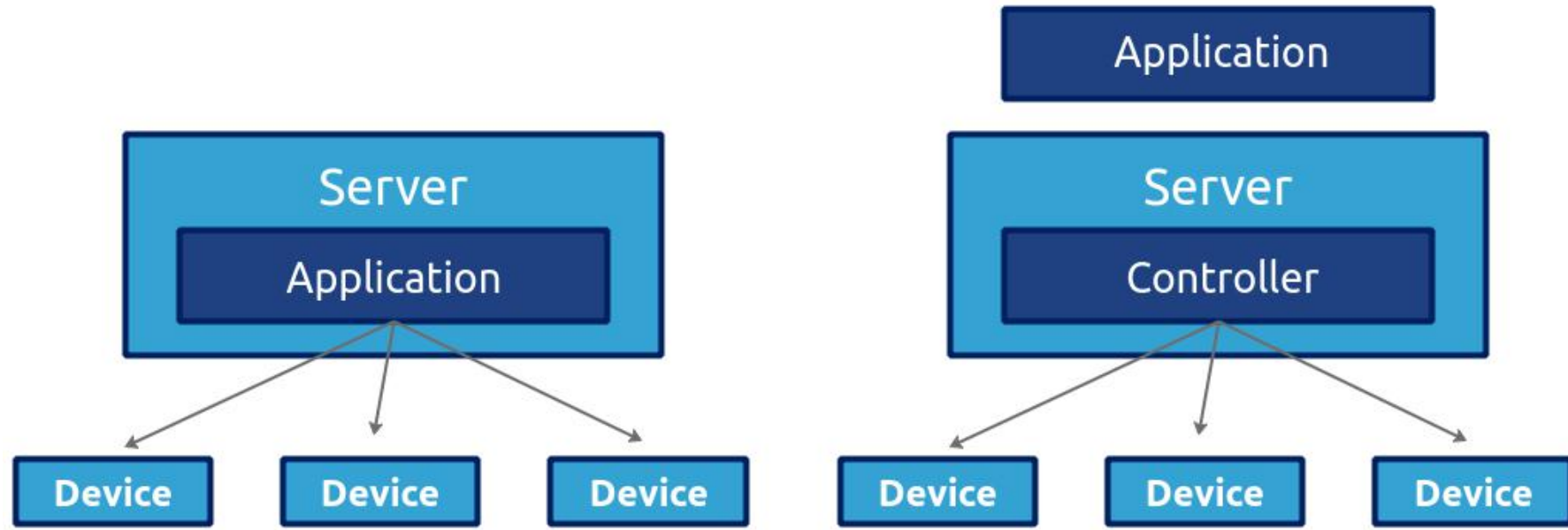
- Support all devices
- Made for human



NETCONF Application

- Dont support all devices
- Made for application

Standalone vs Controller-based Application



Standalone Application

- Application perform discovery, topology, device communication.

Controller-based Application

- Controller perform discovery, topology and device communication.
- Introduction to SDN terminology.



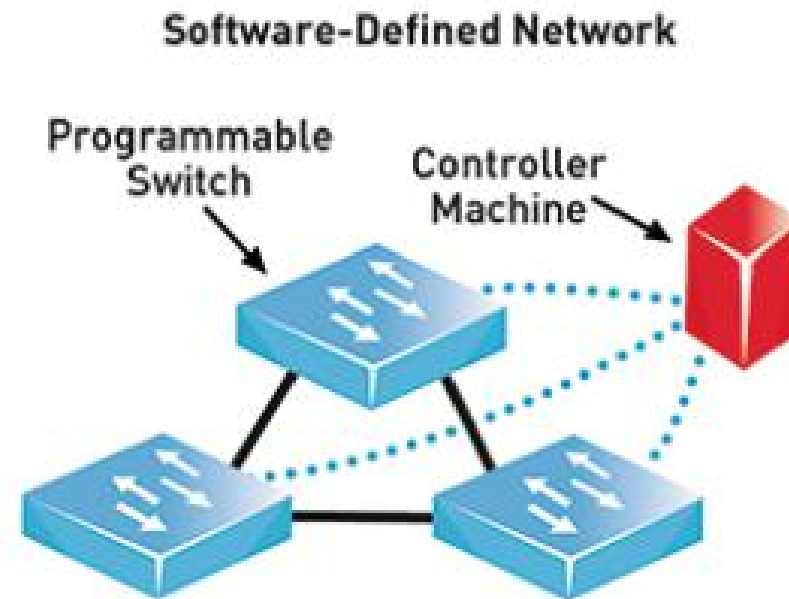
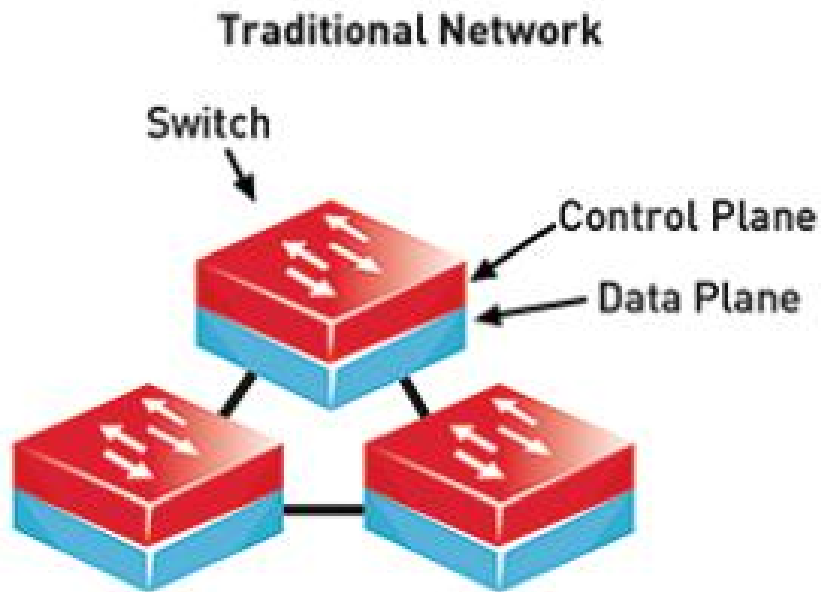
Software Defined Networking

What is SDN?

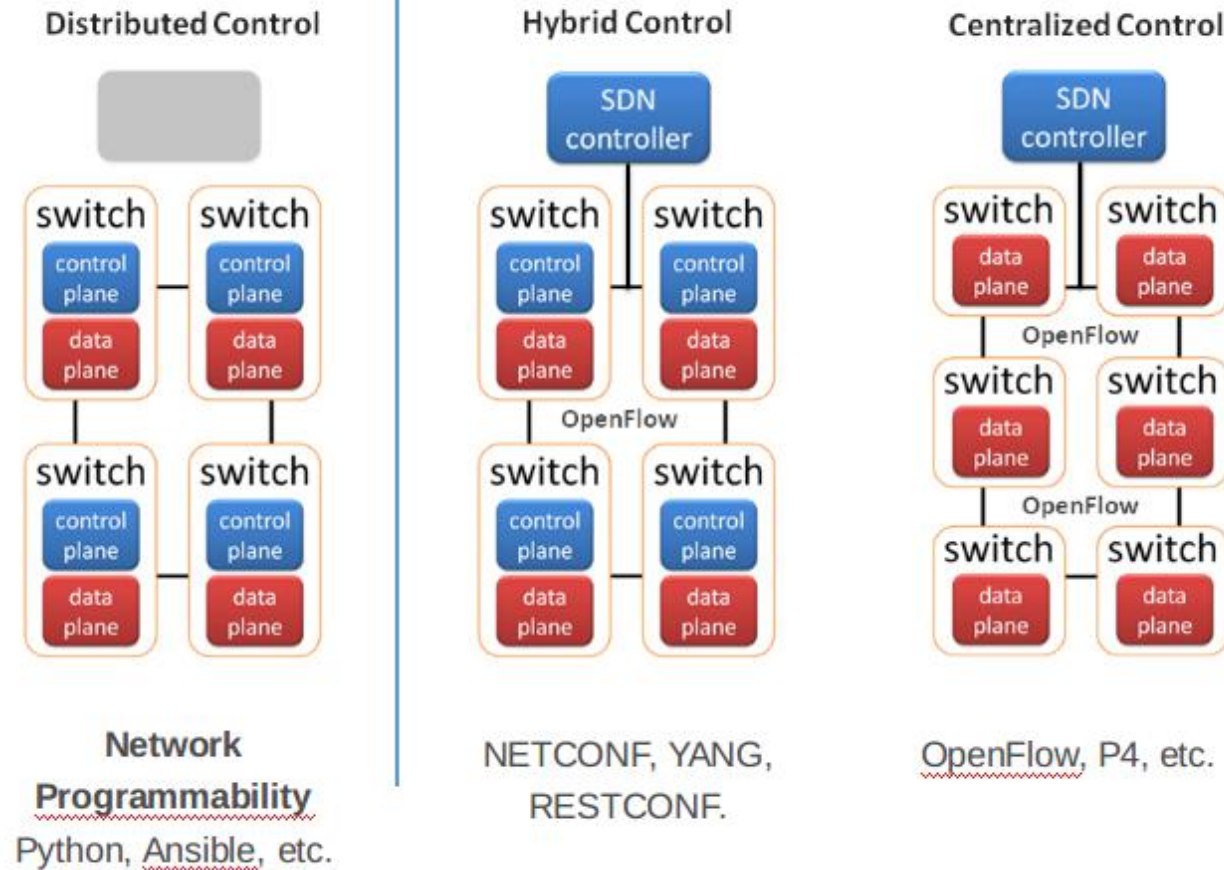
- Separation between Control Plane and Data Plane.
- Deploy Network function into software.
- Abstraction the network via application.

Software Defined Networking refers to a new approach for network programmability, that is, the capacity to initialize, control, change, and manage network behavior dynamically via open interfaces. (RFC 7426)

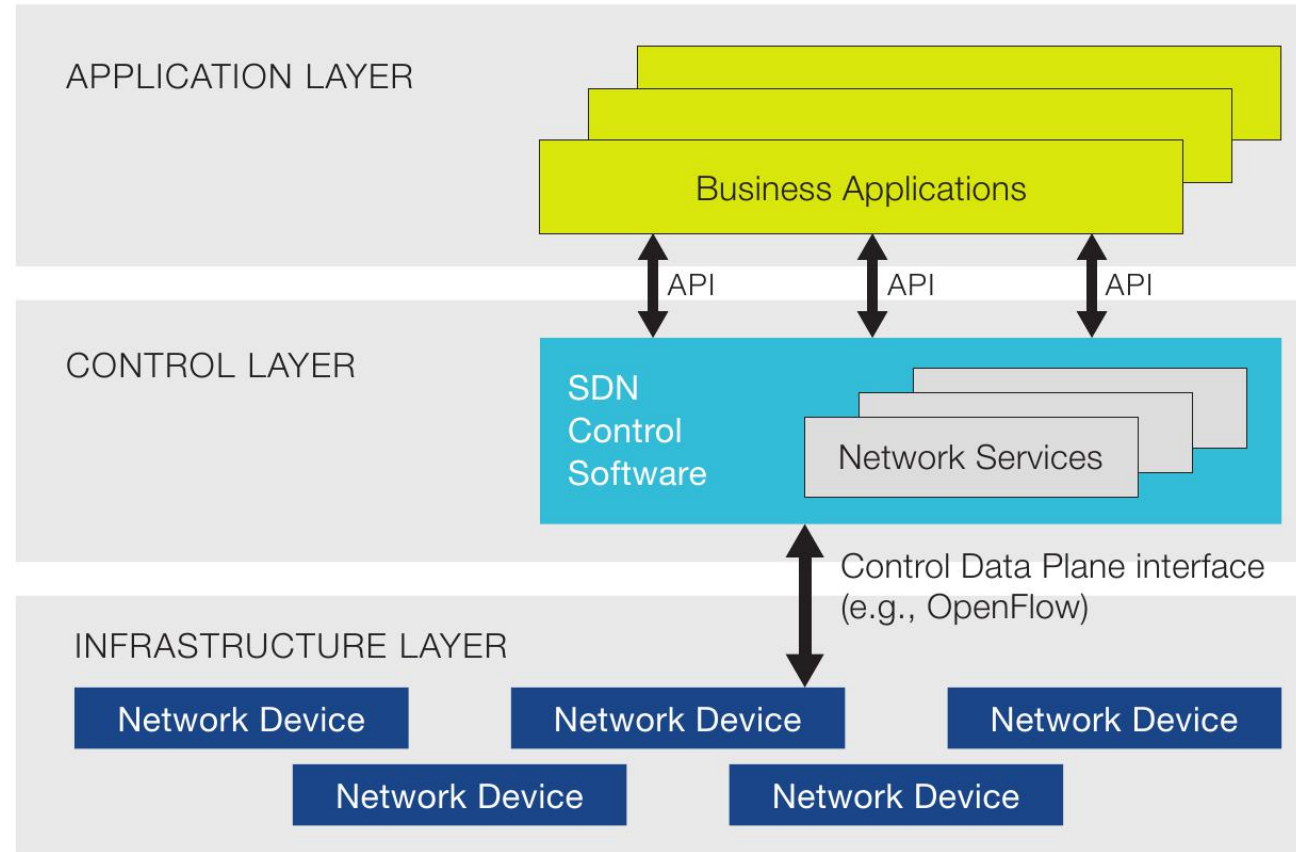
SDN vs Traditional Network



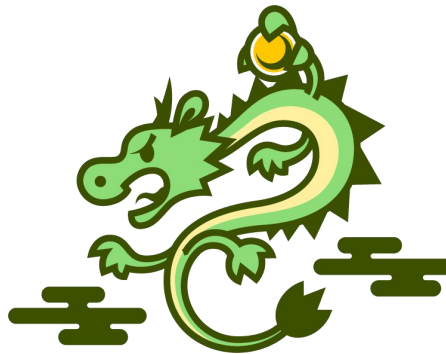
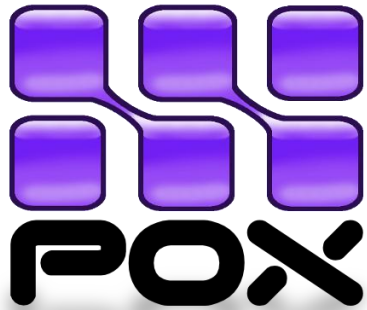
Software Defined Network



SDN Architecture



Controller





Python Overview

Python Overview

- Easy to Learn
- External Libraries
- Large Community
- Networking Examples
- Network Vendors Preferred
- Cross Platform

Python Version

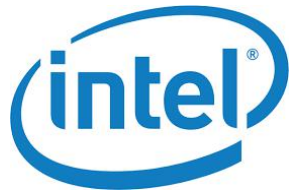
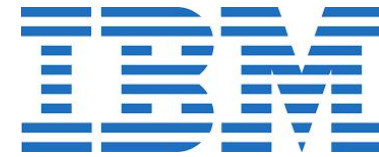
2.x Version

- No longer in development, but support by python community.
- Better library support
- Default on Linux and Mac

3.x Version

- Under Active Development.
- Design to be easier to learn.
- Fix Major issue on 2.x

Python User



Python Installation

Python install by default in ubuntu. If python is not installed, you can install by yourself.

Version 2

```
apt-get install python python-pip
```

Version 3

```
apt-get install python3 python3-pip
```

Modules Install

- There are many modules that is really important to code some program.
- You can install modules manual from command line.
- **pip** for python 2x
- **pip3** for python 3x

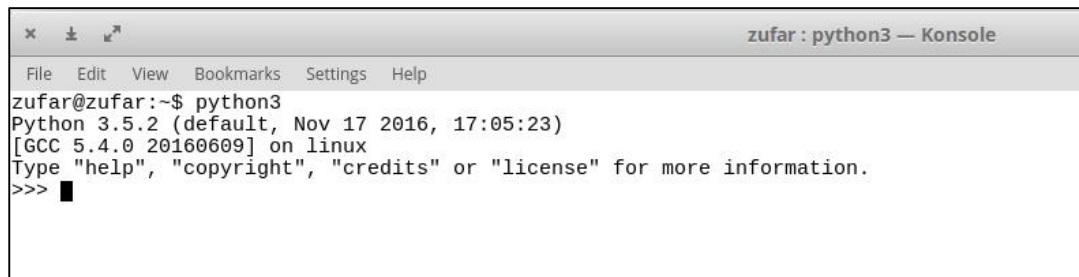
```
# sudo pip install module_name
```

```
# sudo pip3 install module_name
```

Running Python

Using Interpreter

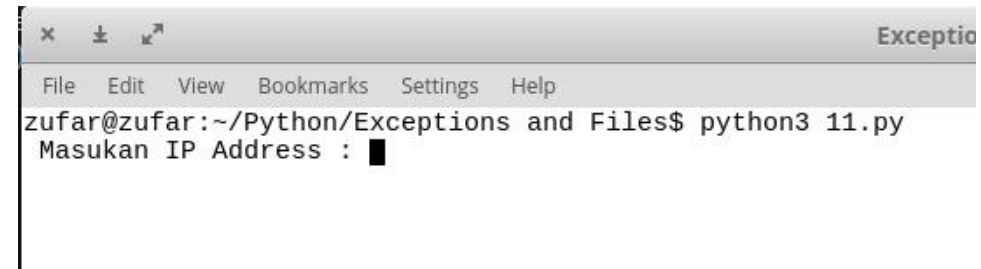
- Just type python or python3 to run the interpreter
- You can enter the code in the interpreter, press enter and execute



```
zufar@zufar:~$ python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

Run the Python File

- You can create the python program in a file and run it.



```
zufar@zufar:~/Python/Exceptions and Files$ python3 11.py
Masukan IP Address : █
```



Reading and Writing Network Device Information

Input/Output Basics

- Open : Before reading and writing into file, file must be open.
- Filename : Must provide the name of the file.
- Mode : Type of operation (**w** for write, **r** for read, and **a** for append).
- Close : When done, be sure to close file.

```
file = open ("file_name","operation_mode"):
    for line in file:
        print (line)
file.close()
```

Writing file

- First, open the file first
- Options to writing file
 - write all line at once with **.write(data)**
 - write a few line with **.writelines(data)**

```
file = open ("file_name","w")  
file.writelines("1.1.1.1\n")  
file.writelines("cisco\n")  
file.writelines("cisco\n")  
file.close()
```


Reading File

- First, open the file first
- Options to reading file
 - write all line at once with `.readlines()`
 - write one line at a time with `.readline()`
 - write file with for looping.

```
file = open ("file_name","r")  
print (file.readlines())  
file.close()
```

```
file = open ("file_name","r")  
print (file.readline())  
print (file.readline())  
file.close()
```



Python and Data Storage

Introduction to Python Storage



Text

- Human readable
- Simple
- Only for testing

Structured
Text

- Human readable
- JSON, XML, CSV
- Conversion to Python data formats

SQL
Database

- Not human readable
- Large volumes of data
- Fast searching capabilities

JSON

- Human Readable
- Easy convert to python data format

Import JSON file

```
database=json.load(open("data.json","r"))
```

Export into JSON

```
json.dump(database, open("data.json","w"))
```



Communicating with Network Devices

Communicating via CLI

- There many tools or modules in python to communicating via CLI. there are Pexpect, Paramiko, and telnetlib.

Pexpect

- General tool for communicating via CLI,
- Manual login process,
- Organized result base on expect() parameters.

Paramiko

- SSH spesific,
- Automatic login process,
- Must parse output manually to match expected result.

Pexpect basic login

- Create session with `.spawn()`
- Organized result base on `.expect()`
- Send command with `.sendline()`

```
session=pexpect.spawn("telnet "+ipaddress,timeout=10)
result=session.expect(["Username: ",pexpect.TIMEOUT])
if result==0:
    session.sendline(username)
    result=session.expect(["Password: ",pexpect.TIMEOUT])
```



Basic Concept Python



Python Data Structures

Data Structure (1)

- Everything you do with computer is managing data.
- Data come in many different shapes
- Python have many built-in Data types

Data Structure (2)

- Numerics : int, long, float, complex, bool.
- Sequences : str, list, tuple, range.
- Mappings : dict.
- Sets : sets, frozenset.

Data Structure (3)

- We do not need to define the data type
- Python Automatically do it for us.
- Check with *type(variable)* in interpreter.

```
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> x=1234
>>> type(x)
<class 'int'>
>>>
>>> y="btech.id"
>>> type(y)
<class 'str'>
>>>
>>> z=["btech.id",123]
>>> type(z)
<class 'list'>
>>> █
```

Mutable vs Immutable Data Types

Mutable

Mutable data type is data that can be changes.

list
set
dict

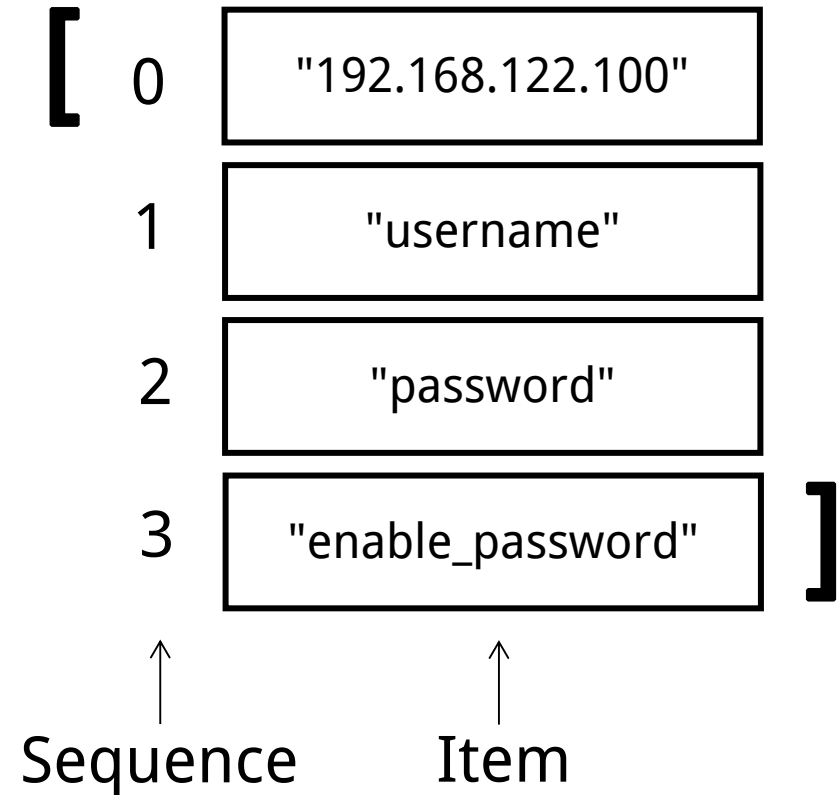
Immutable

Immutable data type is data that can't be changes.

int, float, bool, complex
str
tuple

Overview Lists

- Ordered Sequential list of items
- Item in list can be different types
- be able to add, remove, modify the item.
- items in the list can be data structure



Create List

Create empty list

- Using empty square brackets

```
>>> List = []
```

- Using list() function

```
>>> List = list()
```

Create populated list

```
>>> List = ['192.168.122.100', 'username', 'password']
```

Modify List

- Items inside list can be modify

```
>>> List[0] = '192.168.122.101'
```

- Append a list, items at the end

```
>>> List.append(items)
```

```
>>> List.append("enable_password")
```

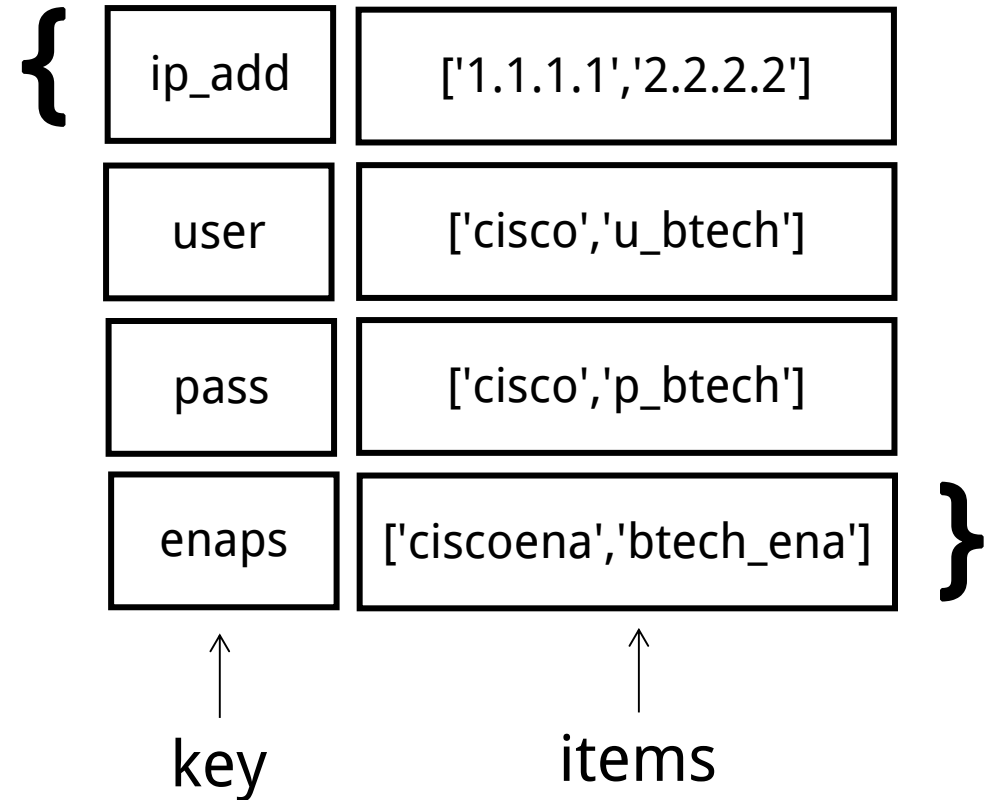
- Insert Items

```
>>> List.insert (sequence,items)
```

```
>>> List.insert (0,"telnet")
```


Overview Dictionaries

- Not sequenced, access by *key*.
- Key must be hashable (string or numeric).
- Items can be added, removed, and changed.
- Items can be simple like string or numeric and complex like list, tuples, and dictionaries.



Create Dictionaries

Create empty dictionaries

- Using empty curly brackets

```
>>> device_data = {}
```

- Using dict() function

```
>>> device_data =dict()
```

Create populated dictionaries

```
>>> device_data = { 'ipadd':['1.1.1.1','2.2.2.2'], 'user':['cisco','u_btech'],  
                    'pass':['cisco','p_btech'], 'enaps':['ciscoena','btech_ena'] }
```

Modify Dictionaries

- Items inside dictionaries can be modify.

```
>>> device_data['ipadd'][0] = '192.168.122.100'
```

- Append a dictionaries, items at the end

```
>>> dictionaries[key].append(items)
```

```
>>> device_data['ipadd'].append('192.168.122.101')
```

- Insert Items

```
>>> dictionaries['key'].insert(sequence,items)
```

```
>>> device_data['ipadd'].insert(0,'123.123.123.1')
```



Conditional Code

Overview

- Conditional code is something we use every day, every moment. The example is : *if the traffic light is green, then i cross; if hungry, eat; if network down, do troubleshoot.*
- Conditional code use **if** as main tool.
- **If** syntax on python

```
if Condition=True:
    program
else:
    program
```
- Use *Operators* to compare condition.

Operators

Comparison Operators : `==, <, >, <=, >=, !=`

Logical Operators : `and, or, not`

Membership Operators : `in, not in`

Identify Operators : `is, is not`

All value is represent in True or False

elif, more condition

- basically, conditional code only match two condition. if we want to match with many condition, we can use **elif** (else if).

- **elif** syntax on python

```
if delay<=10:  
    program  
elif delay >10 and delay <100:  
    program  
else:  
    program
```



Looping

Looping

- being able to execute code block more than once according to the loop parameter we given.
- Looping in python is little bit difference than other language.
- Python have two statement that can be use to do looping, there are for and while statement.

Looping with for and while

for looping

- best using on looping sequence, like list, tuple, dict.
- simple example :

```
for number in [0, 1, 2, 3, 4]:  
    print(number)
```

while looping

- loops as long as a certain condition is True, when the condition get False, the loops end.
- simple example :

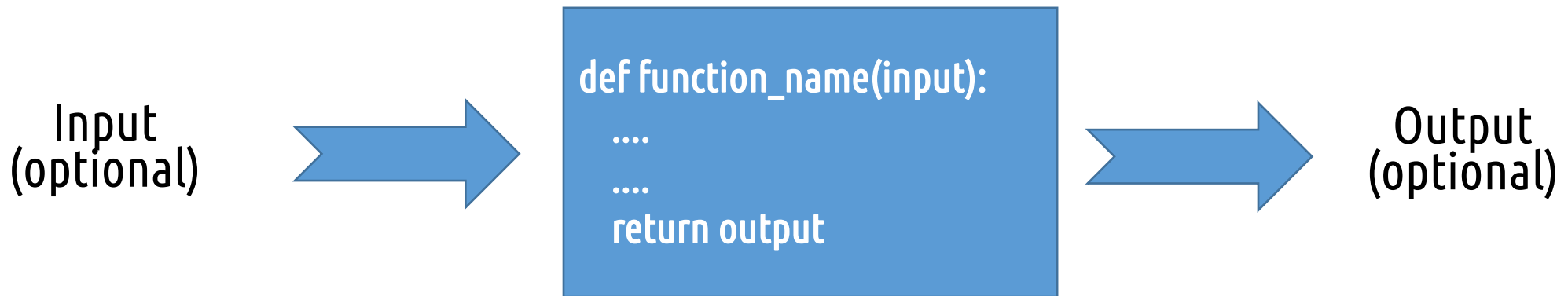
```
while a<=3:  
    print (a)  
    a+=1
```



Functions

functions

- A function is a sequence of instructions that perform a task,
- bundled as a unit. This unit can then be imported and used wherever it's needed.
- Functions can accept input arguments and produce output values. (optional).



Why use functions?

- Reduce Code Duplication, by having a specific task taken care of by a nice block of packaged code that we can import and call whenever we want, we don't need to duplicate its implementation.
- Splitting a complex task or procedure into smaller blocks, each of which becomes a function.
- Improve Readability

Creating functions

- A functions in python is defined by using *def* keyword. after which the name of the function follows, terminated by a pair of braces (which may or may not contain input parameters) and, finally, a colon (:) signals the end of the function definition line.

```
def function_name (input):  
    ....  
    ....  
    return output
```

Calling a functions

```
def function_name(input):  
    ....  
    ....  
    return output
```

```
#main program  
function_name(input)
```

- Define the function in the program.
- Call function in main program.
- Be able to call function more than one.



Introduction to Network APIs and Protocols

Evolution Network Programmability (1)

- Networks were static when protocols such as Simple Network Management Protocol (SNMP) emerged.
- Managing networks via the command line was the norm.
- Network have grown to be overly complex.
- Expect scripting was the norm for those who did do any form of automation.

Evolution Network Programmability (2)

Simple Network Management Protocol

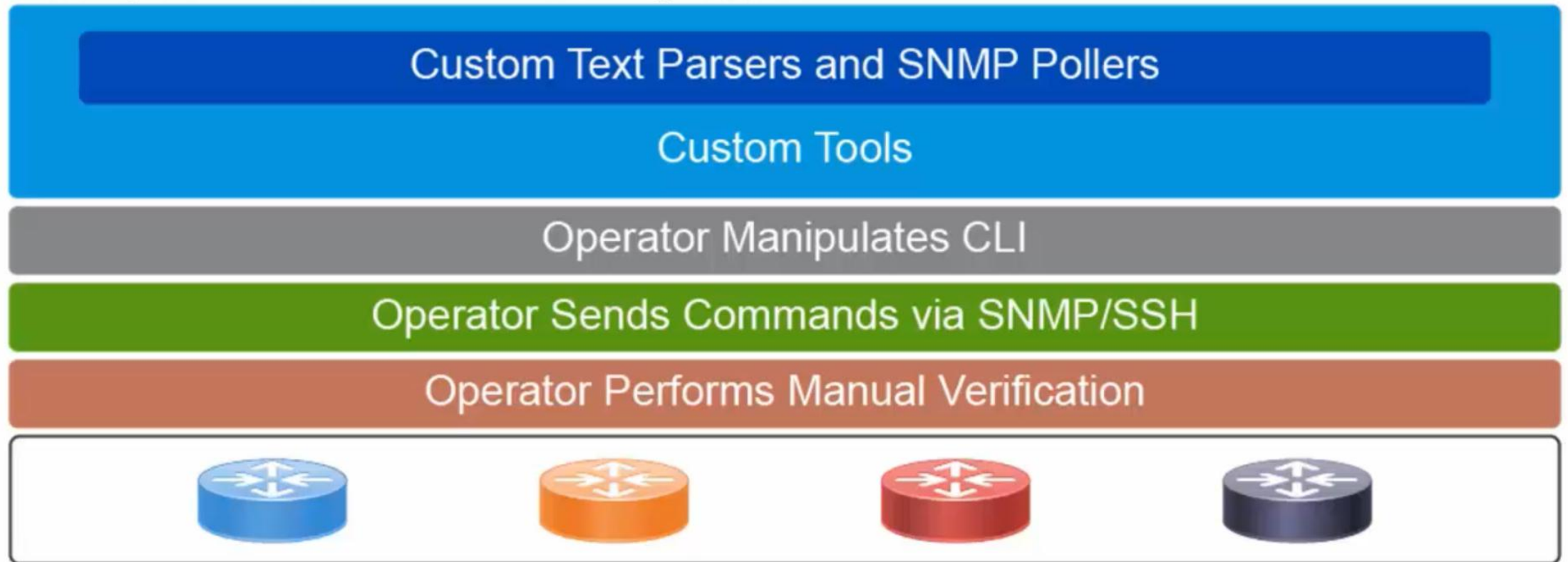
- First developed in late 1980s
- Successfully used for monitoring networks
- SNMP's good when networks were smaller
- Lacks libraries for various programming languages
- Reason why not use SNMP to manage network, RFC 3535.

Evolution Network Programmability (3)

Telnet, SSH, and the Command Line Interface

- Traditional and currently still the primary tools to configuring network devices.
- Requires human interaction
- Not modular, repeatable, or efficient
- All text based - no common data model among platforms

Evolution Network Programmability (4)

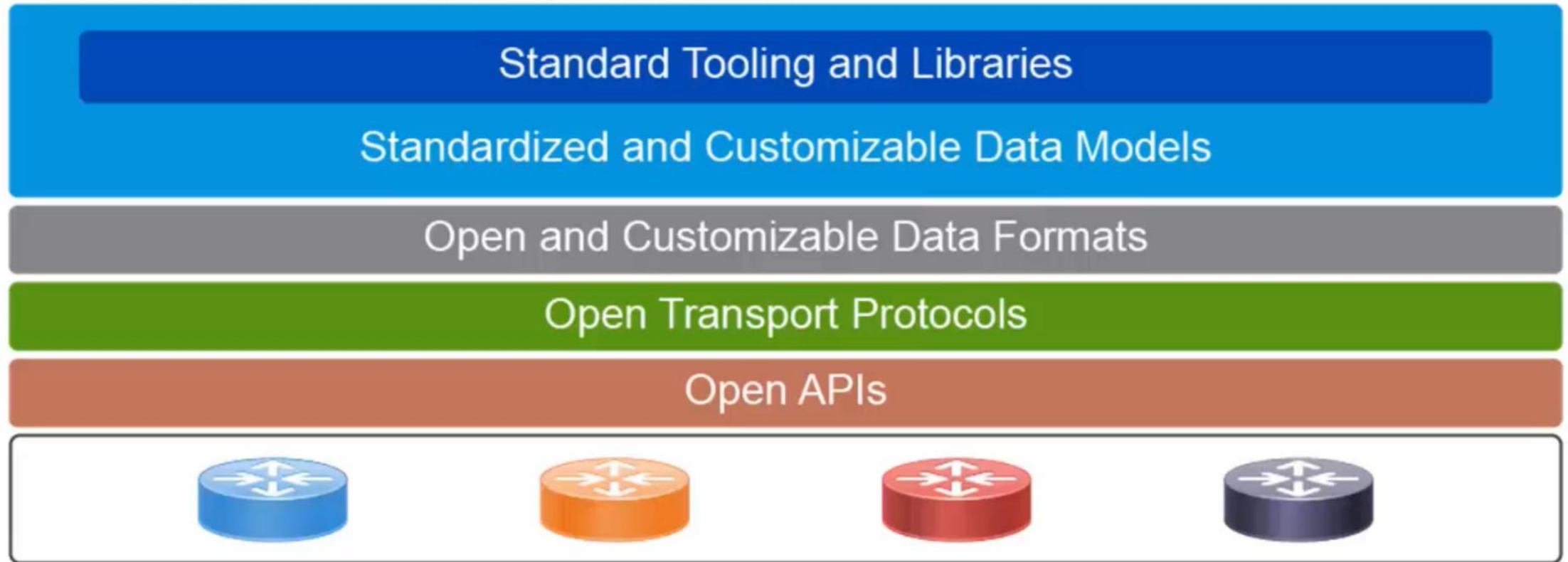


Evolution Network Programmability (5)

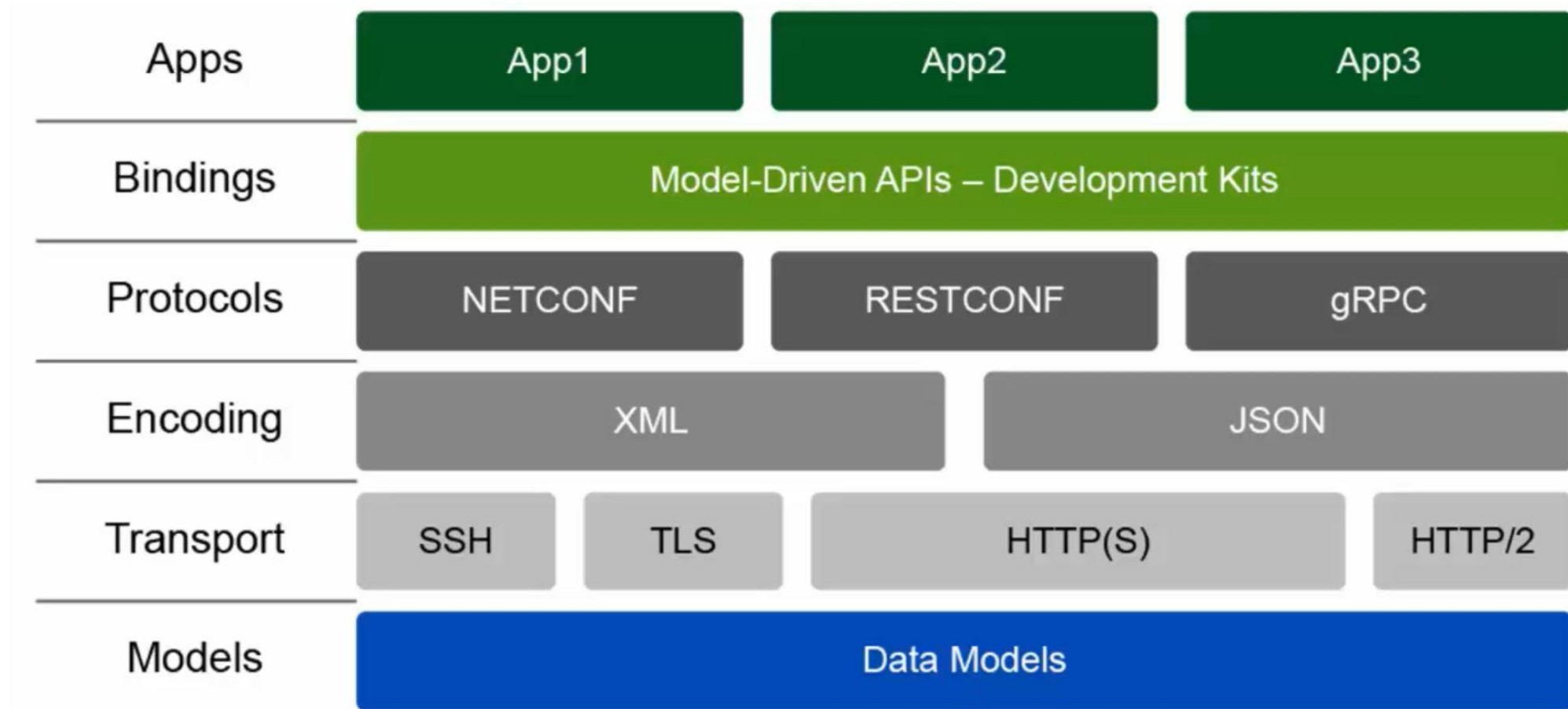
Next Generation Configuration Management Requirements (RFC 3535) :

- Easy to use
- Provides Clear separation between Configuration state and operational state of the device.
- Backup and restore capability.
- Human and Machine friendly.
- Provides Error checking.

Evolution Network Programmability (6)



Model Driven Programmability Stack



JSON

- JavaScript Object Notation.
- Language Independent.
- Way of formatting and transmitting data that is both human and machine readable.
- Sends data objects using name-value pairs
- JSON most closely maps to the dictionary data type in python.
- If you understand Python dictionaries, you understand JSON.

```
{  
    "kind": "object#Version",  
    "selfLink":  
"/api/monitoring/device/version",  
    "firewallMode": "Router",  
    "totalFlashinMB": 8192,  
    "deviceType": "ASAv",  
    "upTimeinSeconds": 21600,  
    "asaVersion": "9.4(1)200",  
    "currentTimeinSeconds": 1508316068  
}
```


XML

- eXtensible Markup Language.
- Language Independent.
- Way of formatting and transmitting data that is both human and machine readable.
- Not as human readable as JSON.
- XML was designed to describe data.
- XML **Tags** are created by the author.

```
<?xml version="1.0" encoding="UTF-8" ?>
  <kind>object#Version</kind>
  <selfLink>/api/monitoring/device/version</selfLink>
  <firewallMode>Router</firewallMode>
  <totalFlashinMB>8192</totalFlashinMB>
  <deviceType>ASAv</deviceType>
  <upTimeinSeconds>21600</upTimeinSeconds>
  <asaVersion>9.4(1)200</asaVersion>
  <currentTimeinSeconds>1508316068</currentTimeinSeconds>
```

Data Models

- Data Model describe a constrained set of data.
- Use well-defined parameters to standardize the representation of data from a network device so the output among various platform is the same.
- Not use to actually send information to the devices, instead rely on protocols such as NETCONF and RESTCONF.
- Device configuration can be validated against a data model in order to check if the changes are a valid for the device before committing the changes.

YANG

- Modeling language defined in RFC 6020
- Initially build for NETCONF
- Now also used by RESTCONF
- Models configuration and operational state data
- Provided syntax and semantics
- Utilizes reusable data structures

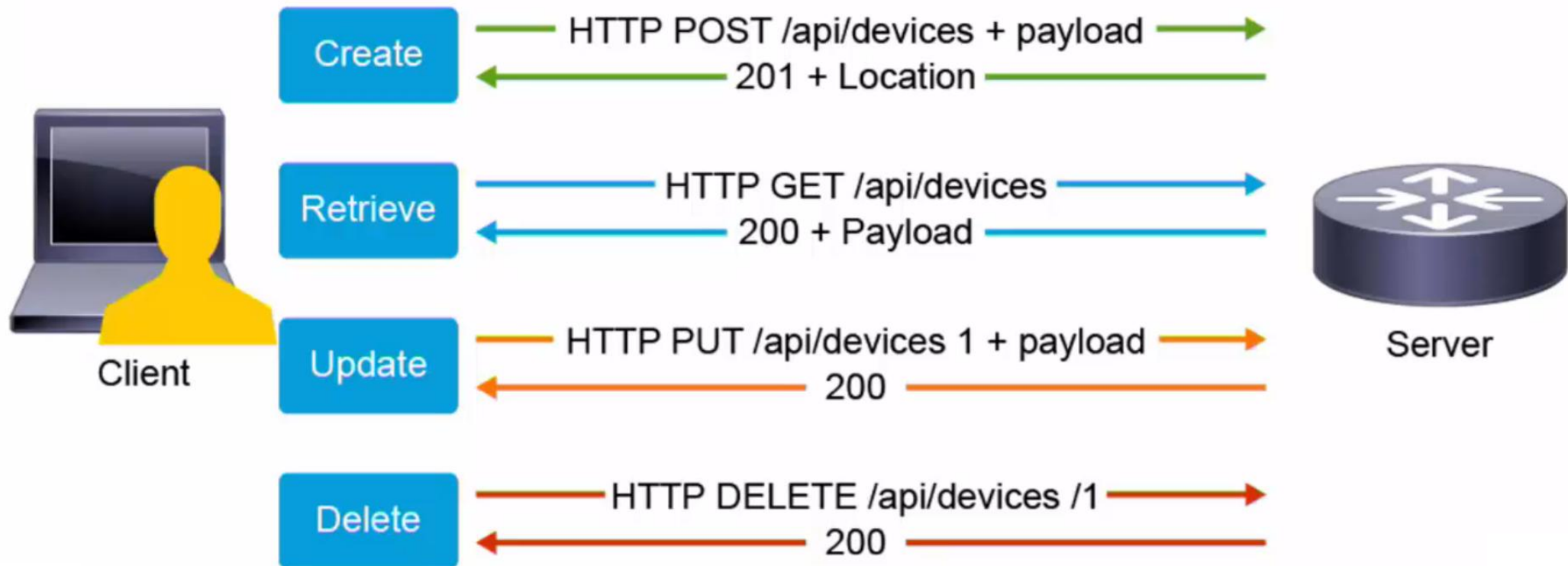
REST (1)

- If you understand how to work with a web browser, you understand REST
- Same HTTP Request methods and response codes are used.



REST (2)

- Create, Retrieve, Update, and Delete (CRUD)



REST (3)

- HTTP Verbs

Get

- Retrieve / Read a resource

Post

- Creates a new resource

Put

- Update / Replace a resource

Patch

- Update / Modify a resource

Delete

- Removes a resource

REST (4)

- HTTP Response Code

Success (2xx)	Description
200	Request Succeeded
201	The request has been fulfilled; new resource created
202	
204	The server fulfilled request but does not return a body

Server Error (5xx)	Description
500	Internal Server Error
501	Not implemented

REST (5)

- HTTP Response Code

Client Error (4xx)	Description
400	Bad Request. Malformed Syntax
401	Unauthorized
403	Server understood request, but refuses to fulfill it
404	Resource not found given URI

REST (6)

- **cURL**

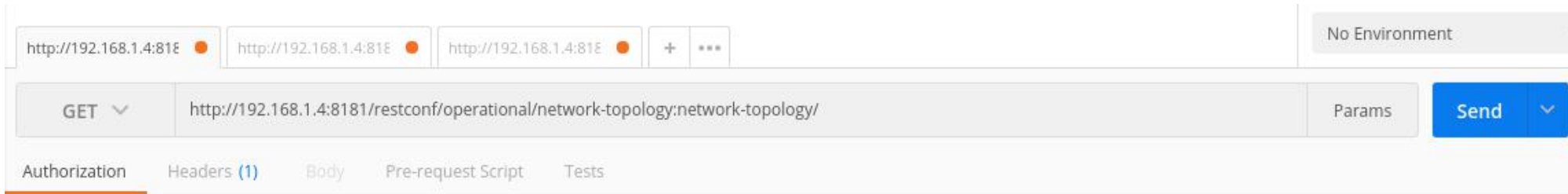
Linux command line tools

- **Postman**

Chrome Application

- **requests**

Python Module



cURL

cURL is a tool to transfer data from or to a server. The command is designed to work without user interaction.

Example :

```
curl -u admin:admin https://192.168.1.2/api/monitoring/clock
```

requests

requests is a tool in python that use to send data to the server.

Example :

```
url = 'https://192.168.1.1/api/monitoring/device/components/version'  
response = requests.get(url,auth=('', ''),verify=False)  
print response
```

NETCONF (1)

NETCONF is a IETF network management protocol designed specifically for configuration management

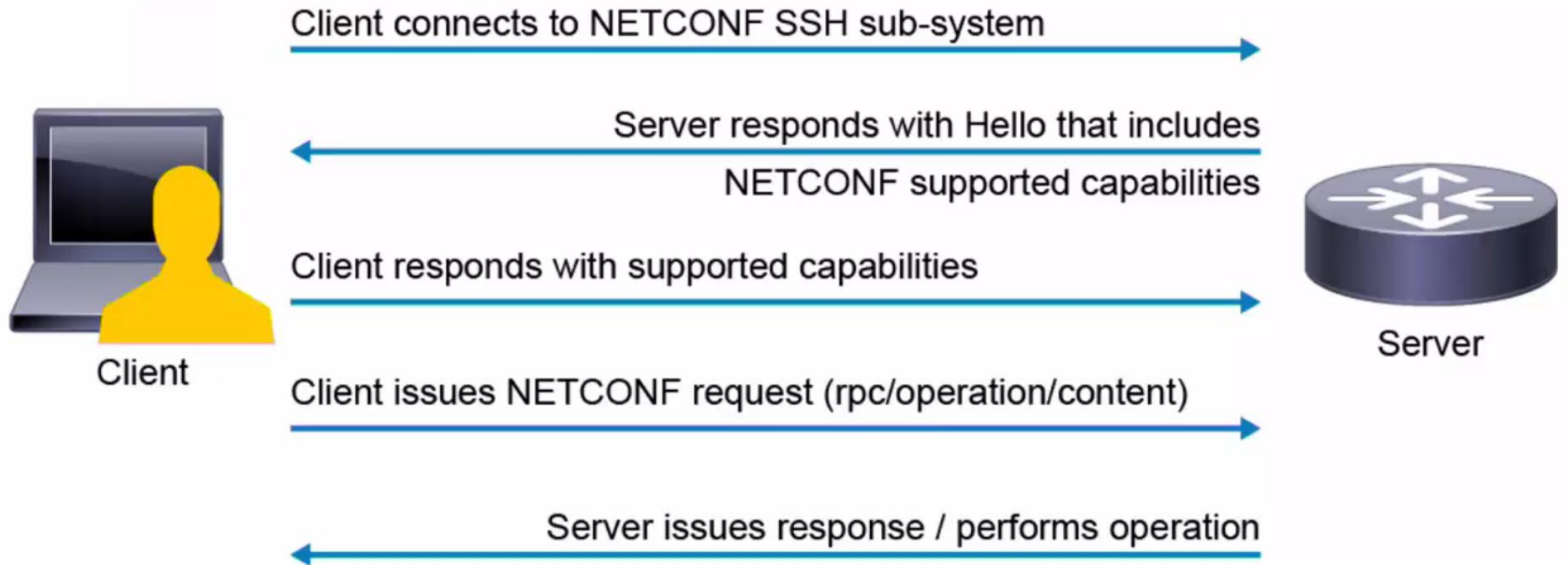
- Makes a distinction between configuration and state data.
- Utilized multiple configuration data stores (candidate, running, startup).
- Configuration change transactions.
- Provides client side configuration validation.
- Uses a client-server model and SSH as transport protocol

NETCONF (2)

Protocol Stack

Layer	Example
Protocols	SSH
Messages	<rpc>,<rpc-reply>
Operations	<get-config>,<get>,<copy-config>,<commit>,<validate>,<edit-config>,<delete-config>
Content	XML Document

NETCONF (3)



NETCONF (4)

- ncclient

ncclient is a Python library for NETCONF clients.

```
with manager.connect(host=host, port=port, username=user, password=pwd,  
hostkey_verify=False, device_params={'name': 'csr'}) as m:  
    capabilities = m.server_capabilities
```

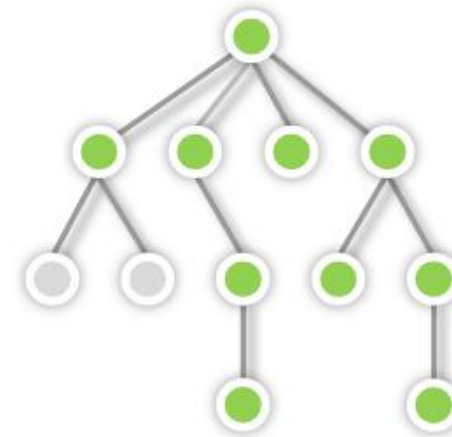


YANG Data Modeling and Tools

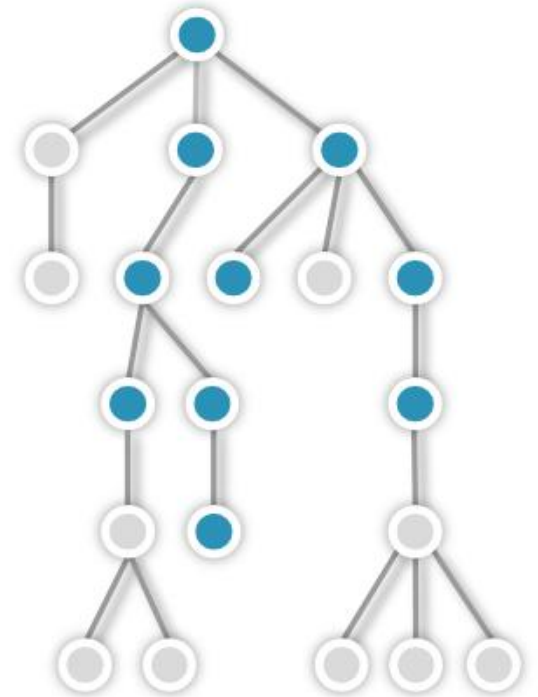
Overview

- YANG is a modeling language defined in RFC 6020.
- Models files are self-documented and ship with devices
- The goal is to be able to formally model any network device configuration.
- YANG data models is create by IETF or OpenConfig and mapped to native models in each vendors.

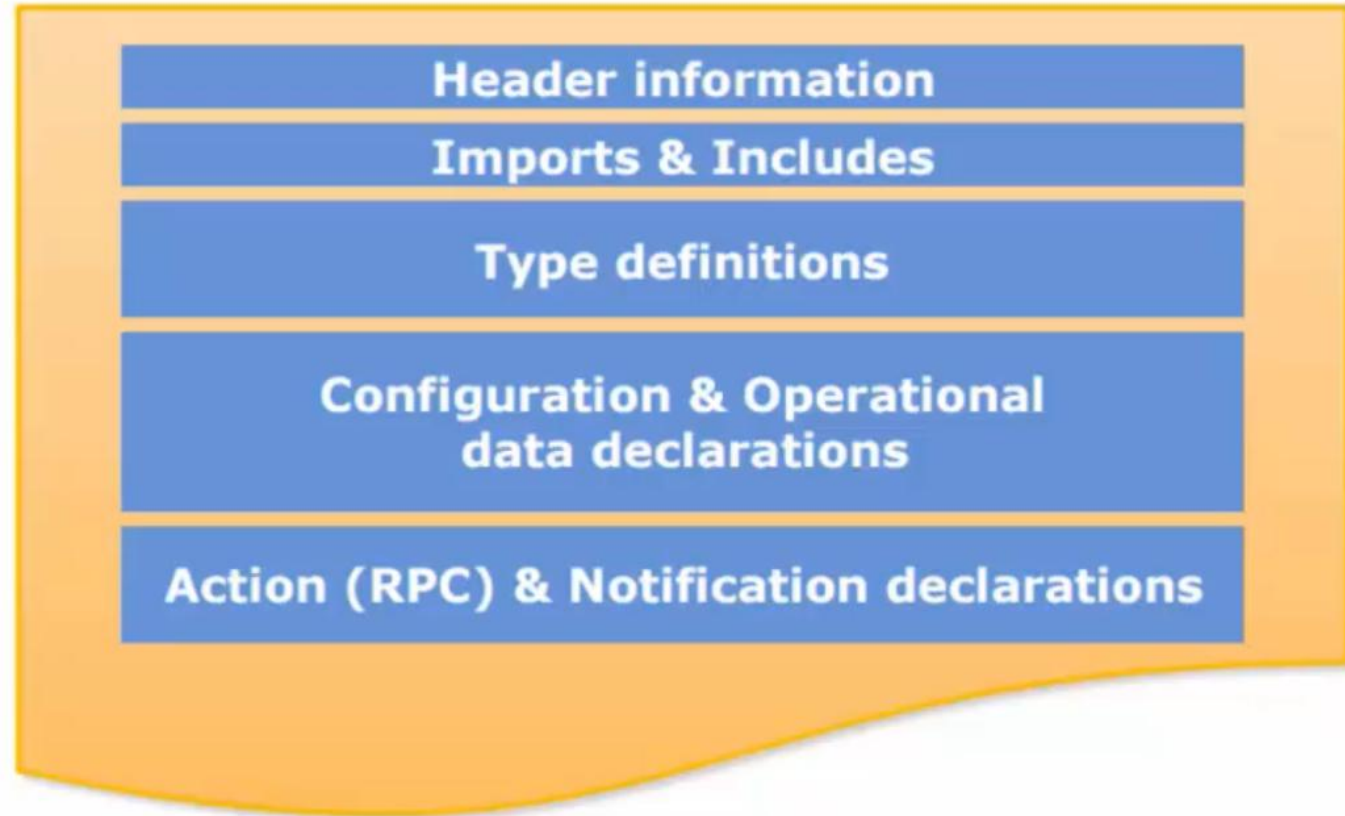
OpenConfig / IETF



Native / Common



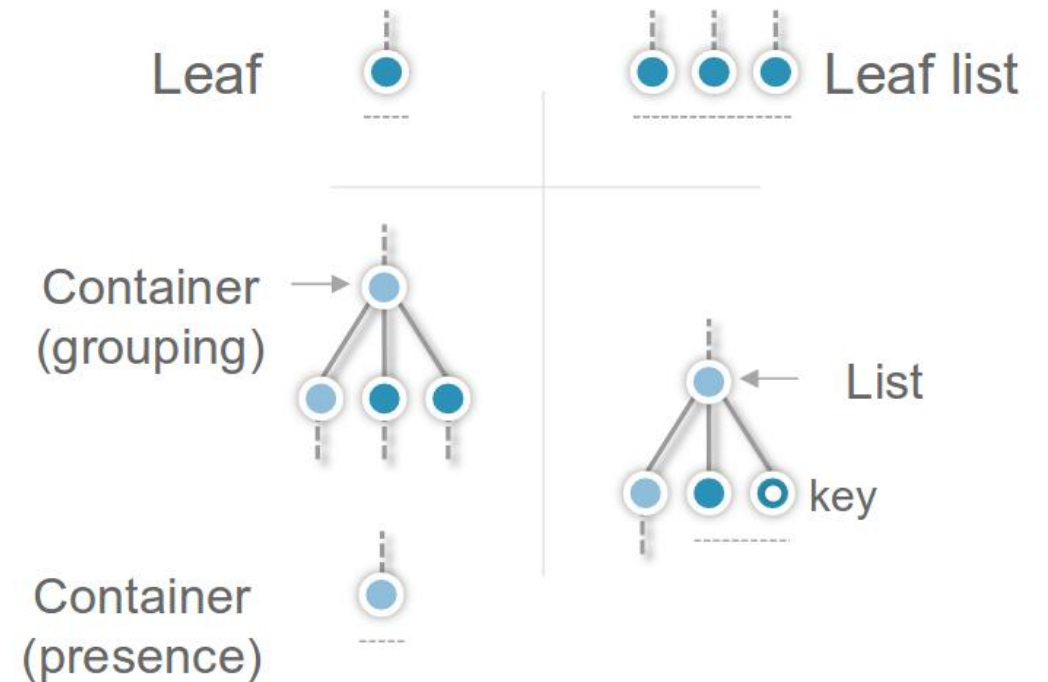
YANG Module



Overview

Main node types :

- Leaf – node with name and value of certain type (no children)
- Leaf list – sequence of leafs
- Container – groups nodes and has no value
- List – Sequence of records with key leafs



Leaf Node

- Contains simple data like an integer or a string.
- It has exactly one value of a particular type.
- No child nodes
- In JSON leaf is mapped to a single key/value pair 'name': 'value'

```
leaf host-name {  
    type string;  
    description "Hostname for this  
system";  
}
```

NETCONF XML Example:

```
<host-name>my.example.com</host-  
name>
```

Leaf-List Node

- A leaf-list is a sequence of leaf nodes with exactly one value of a particular type per leaf.

```
leaf-list domain-search {  
    type string;  
    description "List of domain names to  
    search";  
}
```

NETCONF XML Example:

```
<domain-  
search>high.example.com</domain-search>  
<domain-search>low.example.com</domain-  
search>
```

Container Node

- A container node is used to group related nodes in a subtree.
- A container has only child nodes and no value.
- A container may contain any number of child nodes of any type (including leafs, lists, containers, and leaf-lists).
- In JSON it is mapped to a name/object pair 'name': {...}

```
container system {  
    container login {  
        leaf message {  
            type string;  
            description  
            "Message given at start of login  
            session";  
        }  
    }  
}
```

List Node

- A list defines a sequence of list entries.
- Each entry is like a structure or a record instance, and is uniquely identified by the values of its key leafs.
- A list can define multiple key leafs and may contain any number of child nodes of any type (including leafs, lists, containers etc.).
- In JSON lists are encoded as name/arrays pairs containing JSON objects 'name': [{...}, {...}]

```
list user {  
    key "name";  
    leaf name {  
        type string;  
    }  
    leaf full-name {  
        type string;  
    }  
    leaf class {  
        type string;  
    }  
}
```

Configuration and State Data

YANG can model state data, as well as configuration data, based on the "config" statement. When a node is tagged with "config false", its subhierarchy is flagged as state data. By default, all YANG syntax are configuration data.

Yang Types (1)

- RFC 6021
- Most YANG elements have a data type.
- Type may be a base type or derived type :
 - Derived types may be simple typedefs or groupings.
 - There are 20+ base types to start

Type	Description
int8/16/32/64	Integer
uint8/16/32/64	Unsigned integer
decimal64	Non-integer
string	Unicode string
enumeration	Set of alternatives
boolean	True or false
bits	Boolean array
binary	Binary BLOB
leafref	Reference "pointer"

Yang Types (2)

Common YANG Types

- Commonly used YANG types is defined in RFC 6021
- Use :

```
import "ietf-yang-types" {  
    prefix yang;  
}
```

counter32/64	ipv4-address
gauge32/64	ipv6-address
object-identifier	ip-prefix
date-and-time	ipv4-prefix
timeticks	ipv6-prefix
timestamp	domain-name
phys-address	uri
ip-version	mac-address
flow-label	bridgeid
port-number	vlanid
ip-address	...and more

Derived Types (typedef)

YANG can define derived types from base types using the **typedef** statement.

A base type can be either a built-in type or a derived type, allowing a hierarchy of derived types.

```
typedef percent {  
    type uint8 {  
        range "0 .. 100";  
    }  
    description "Percentage";  
}  
  
leaf completed {  
    type percent;  
}
```

Type Restrictions

Type Restrictions allow YANG to restrict a type to adjust a network environment.


```
typedef base-int32 {  
    type int32 {  
        range "1..4 | 10..20";  
    }  
}
```

```
typedef derived-int32 {  
    type base-int32 {  
        range "11..max";  
    }  
}
```

Reusable Node Groups (grouping)

Groups of nodes can be assembled into reusable collections using the **grouping** statement. A grouping defines a set of nodes that are instantiated with the **uses** statement

```
grouping target {  
    leaf address {  
        type inet:ip-address;  
        description "Target IP address";  
    }  
    leaf port {  
        type inet:port-number;  
        description "Target port number";  
    }  
}  
  
container peer {  
    container destination {  
        uses target;  
    }  
}
```



<https://btech.id>