
Automating IBM z/OS Mainframe Systems with Python Scripts

Methods, Use Cases, and Best Practices for Modernizing Mainframe Automation

Diego Rodríguez Bravo & Tom McQuitty Jan. 2025

Contents

Introduction	3
Baseline concepts	4
Use Case description	4
Target data	4
REXX & z/OS	5
Python & USS	7
Executing Python scripts	10
Batch	10
REXX	11
OMVS	11
ISPF 3.17	12
ZOAU	13
Python & ZOAU	14
Python & ZOAU APIs	17
Remote connection	20
SSH	20
Python & SSH	22
Python & SSH & ZOAU Explicit Environment	24
Python & SSH & ZOAU API Login shell	26
z/OSMF	29
Python & z/OSMF	30
Zowe	32
Zowe API ML	32
Python & Zowe API ML	33
Python & Zowe CLI	35
Python & zowe SDK	37
Samples	39
IPL Map USS ZOAU Python APIs	40
IPL Map Launcher SSH & Zowe	53
GitHub repository	59

Introduction

Python scripts have emerged as a powerful tool for automating tasks in IBM z/OS environments, offering significant improvements in efficiency, flexibility, and scalability. Leveraging Python's extensive libraries and robust capabilities, users can simplify complex workflows, reduce manual intervention, and enhance system reliability.

This paper explores the utility of Python in automating z/OS tasks such as data manipulation, system monitoring, job scheduling, and file management. It highlights the integration of Python with z/OS Open Automation Utilities (ZOAU) and IBM's z/OS Management Facility (z/OSMF), leveraging REST APIs to streamline operations and improve response times. Additionally, the use of Python to bridge traditional z/OS processes with modern DevOps practices is examined, showcasing its adaptability in CI/CD pipelines, log analysis, and system diagnostics. Scenarios such as migrating legacy applications, managing large datasets, and orchestrating multi-system processes are discussed to illustrate the versatility of Python automation.

By adopting Python scripts, organizations can unlock the potential for cost savings, error reduction, and accelerated innovation, demonstrating its value across a variety of use cases and operational circumstances. With a significant number of developers working with Python, the pool of available developers for mainframe applications is increasing.

This document assumes that you're familiarized with some concepts, like Python, Zowe, ZOAU and z/OSMF and the intention is to show different ways to obtain the same task done via different ways. There are multiple approaches for each solution. Your approach will depend on your goals, skills, company policies, teamwork, plans to maintain and grow, etc.

Baseline concepts

Use Case description

For demonstration purposes, the use case will list the members in a partitioned dataset and retrieve the content of one of them via different methods. Accessing datasets or members is a common practice since they can contain log information, input records for processing, parameters, queries, code or any other data that would apply for an automation use case. We will be using Python version 3.x. We will use scripts in different environments and use different resources and frameworks to achieve our goals.

Target data

There is a Partitioned Data Set in our mainframe named: **PROD001.TENNIS** with members for countries. Each member stores a list of tennis players:

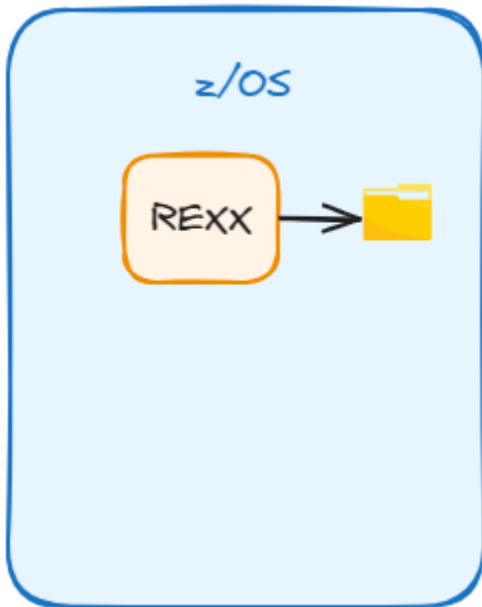
```
VIEW          PROD001.TENNIS          Row 0000001 of 0000003
Command ==>          Scroll ==> CSR
      Name      Prompt      Size  Created      Changed      ID
-----  FRANCE          5  2025/04/09  2025/04/09 14:13:52  PROD001
-----  SERBIA          5  2025/04/09  2025/04/09 14:14:18  PROD001
-----  USA            5  2025/04/09  2025/04/09 14:12:06  PROD001
**End**
```

```
VIEW          PROD001.TENNIS(USA) - 01.01          Columns 00001 00080
Command ==>          Scroll ==> CSR
***** ***** Top of Data *****
000001 Serena Williams
000002 Venus Williams
000003 Pete Sampras
000004 Andre Agassi
000005 Andy Roddick
***** ***** Bottom of Data *****
```

After running our script, we will have:

1. List with all the member names - Brands
2. List with the content of the first member – France players

REXX & z/OS



To establish a baseline situation, here is our sample application coded in REXX. There are a few ways to achieve our goal, like using ISPF service 'LMMLIST' or directly reading the file directory. This sample utilizes TSO command 'LISTDS':

```
/* REXX *****  
* - Retrieve all member names from a PDS: PROD001.TENNIS      *  
* - Retrieve the content of the first member                  *  
*****  
pds = 'PROD001.TENNIS'  
  
x = outtrap(list.)  
"listds '"pds"' members"  
x = outtrap(off)  
say pds; say copies('-',25)  
start='no'  
cont = 0  
do i = 0 to list.0  
    list.i=strip(list.i)  
    if start = 'yes' then do  
        cont=cont+1  
        members.cont=list.i  
        say members.cont  
    end  
    if list.i = '--MEMBERS--' then start='yes'  
end
```

```
members.0=cont

"alloc da('"pds"("members.1")') fi(mydd) shr reu"
"execio * disk mydd (finis stem frstmem."; "free fi(mydd)"
say ' '; say pds("members.1"); say copies('-',25)
do i = 1 to frstmem.0
    say frstmem.i
end
exit
```

Here is the output:

PROD001.TENNIS

FRANCE

SERBIA

USA

PROD001.TENNIS(FRANCE)

Yannick Noah

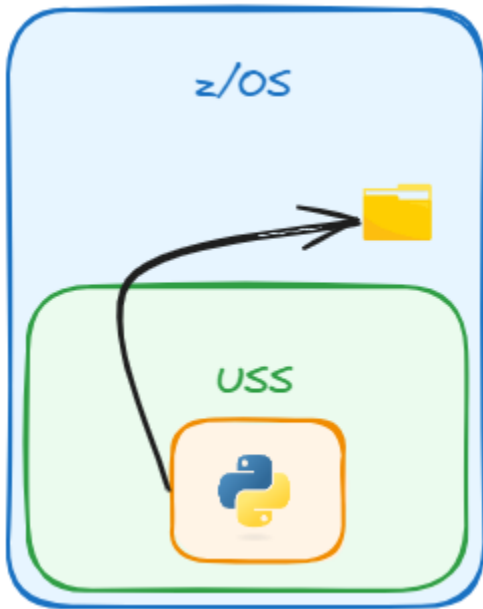
Jo-Wilfried Tsonga

Amelie Mauresmo

Gael Monfils

Marion Bartoli

Python & USS



The following examples are used for basic understanding and not the most pythonic examples. The intention is to be simple. The code is not robust, and the goal is to simplify the process.

Python code structure usually looks like the following:

```
# import libraries section
import ...
from ... import ..., ...

# files management
with open...

# function
def do_something():
    ...
    return someting

# run
if __name__ == "__main__":
    data = do_something() # call function
    print('data: ', data)
```

💡 Note: Python supports libraries. They are added to the project using the import command. Libraries solve common problems, such as accessing files, various APIs, etc. In this example, the

subprocess library is used to access the command shell to execute commands, but later examples will show calling MVS resources using a library, as well.

To take advantage of using shell commands we will use a common Python snippet to execute available system commands. Here is an example program flow to act in case the command does not run as expected:

```
import subprocess

def execute_command(command):
    try:
        result = subprocess.run(command,
                                shell=True, text=True, capture_output=True)
    except subprocess.CalledProcessError as e:
        print("Command execution failed.")
        print("Return Code:", e.returncode)
        print("Error:", e.stderr)
    return result.stdout, result.stderr, result.returncode

sto, ste, rc = execute_command('<command parms flags ...>')
```

This is a function. We pass in the command we want to run, and it collects the output and returns the results. The first line imports a library, in this case 'subprocess'. Subprocess allows us to run command line utilities. It executes the command and returns the output, which is stored in the `sto` (standard output), `ste` (standard error) and `rc` (return code) variables.

In this case we will use the '`tsocmd`' Shell command that allows us to execute any TSO command in the same way we used in the previous REXX sample by calling the TSO '`listds`' facility. Then we will use the '`cat`' Shell command to print the first member. The code will look something like:

```
# Python script in USS -----
# - Retrieve all member names from a PDS: PROD001.TENNIS
# - Retrieve the content of the first member
#-----
import subprocess

def execute_command(command):
    try:
        result = subprocess.run(command,
                                shell=True, text=True, capture_output=True)
    except subprocess.CalledProcessError as e:
        print("Command execution failed.")
        print("Return Code:", e.returncode)
        print("Error:", e.stderr)
    return result.stdout, result.stderr, result.returncode
```




```

pds = 'PROD001.TENNIS'

members, ste, rc = execute_command(f'tsocmd "listds \'{pds}\''
members"')
members = members.split("--MEMBERS--")[1]
members = [x.strip() for x in members.split('\n')]
members[:] = [item for item in members if item != ""]
print(f'{pds}')
print('-'*25)
for member in members:
    print(member)

firstmem = members[0]
file_content, ste, rc = execute_command(f'cat
"/\'/{pds}({firstmem})\'"')
print(f'{pds}({firstmem})')
print('-'*25)
print(file_content)

```

 Tip: When the solution has already been written in z/OS, we can call those applications directly. We could just call our REXX script from our previous example with the 'tsocmd' command:


```

# call rexx exec

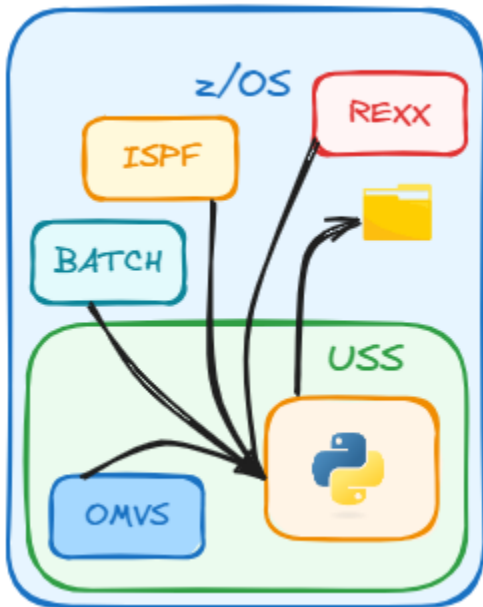
myrex = 'mylib.rexx(sample1)'

data, ste, rc = execute_command(f'tsocmd "ex \'{myrex}\''
data = data.split('\n')
for line in data:
    print(line)

```

 Warning: The output data (members list and the content of the first one) will reside in the same variable (data) so we will need to parse it and proceed as we need. In this case, it would make much more sense to work with arguments, to provide the PDS name and the member we want to display, by position or directly by name.

Executing Python scripts



There are multiple ways to execute the script, including batch, event triggered, scheduled or even as a service.

Here are some examples:

Batch

'BPXBATCH' can execute a Python script in USS via JCL. Python runtime can be referenced via PARM if needed. The script is addressed via STDPARM DD or in the EXEC card. Here are both samples:

```
//PYTBATCH JOB (124400000), 'MYDEPT', CLASS=A, MSGCLASS=X,  
//          NOTIFY=&SYSUID  
//STEP1     EXEC PGM=BPXBATCH, PARM='SH python /u/users/group/product/pro  
//          d001/python-mainframe/myScript.py'  
//STDERR    DD SYSOUT=*  
//STDOUT    DD SYSOUT=*  
/*
```

```
//PYTBATCH JOB (124400000), 'MYDEPT', CLASS=A, MSGCLASS=X,  
//          NOTIFY=&SYSUID  
//STEP1     EXEC PGM=BPXBATCH  
//STDERR    DD SYSOUT=*  
//STDOUT    DD SYSOUT=*  
//STDPARM   DD *
```

```
sh python /u/users/user01/python-mainframe/myScript.py
/*
```

REXX

'BPXUNIX' executes commands in USS. Basic shell commands like 'ls', 'cat' or 'pwd' can be run without any settings, but for commands that need some environment variables access like the userid default login options or set new variables for PATH or LIBPATH need to be set. In this case, the home directory path is set first and set it in the 'env' stem as HOME so I can login to execute Python.

```
>>-bpxwunix--(--cmd--+-----+--)-><
                        '-,stdin--+-----+-'
                                '-,stdout--+-----+-'
                                        '-,stderr--+-----+-'
                                                '-,env--+-----+-'
                                                        '-,login--+-----+-'
                                                                '-,batch--'
```

```
/* REXX *****
* call python script
***** /
cmd = "pwd"
x=bpxwunix(cmd,,std.,err.)
env.0 = 1
env.1='HOME=' || std.1

dir = './python-mainframe/'
scr = 'myScript.py'
cmd = 'python 'dir||scr
x=bpxwunix(cmd,,std.,err.,env.,1)
do i = 1 to std.0; say 'std.' std.i; end
exit
```

OMVS

TSO OMVS session within your 3270 emulation (or puTTY, ssh, Telnet). Position yourself in the Python scripts directory and execute:

```
# python myScript.py
```

ISPF 3.17

From 3.17 - Udlis option in ISPF. We need to provide the whole path or the relative from our \$HOME

Command ==> `python ./python-mainframe/myScript.py`

Draft

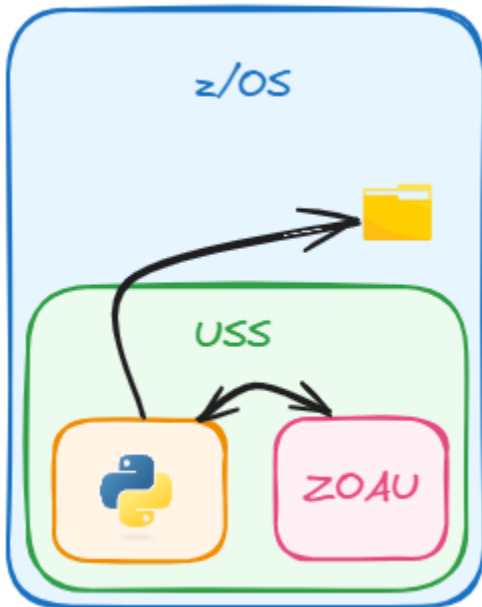
ZOAU

ZOAU stands for IBM Z Open Automation Utilities.

Everyday the number of automation and program management scripts is growing within the USS (UNIX System Services) environments, sometimes being the preferred choice instead of JCLs. But in certain situations, the scripts to automate tasks in USS environment are not the best solution due a lack of z/OS facilities interaction. To fill this gap, there is ZOAU, a suite of command line utilities of z/OS Unix System Services that can manage z/OS facilities by providing a wide set of utilities meant for MVS program execution, Data Set manipulation and management, JES interaction and SYSPROG and operator commands.

Embedding ZOAU into modern scripting languages like Python leverages a functionality enough to achieve almost any possible action within our Mainframe infrastructure.

Python & ZOAU



This use case uses ZOAU commands to list the member list with the 'mls' command, and for reading the first member we will use the 'dcat' command.

```
# Python script in USS using ZOAU -----
# - Retrieve all member names from a PDS: PROD001.TENNIS
# - Retrieve the content of the first member
#-----
import subprocess

def execute_command(command):
    try:
        result = subprocess.run(command,
                                shell=True, text=True, capture_output=True)
    except subprocess.CalledProcessError as e:
        print("Command execution failed.")
        print("Return Code:", e.returncode)
        print("Error:", e.stderr)
    return result.stdout, result.stderr, result.returncode

pds = 'PROD001.TENNIS'

members, ste, rc = execute_command(f'mls "{pds}"')
members = [x.strip() for x in members.split('\n')]
print(f'{pds}')
```

```

print('-'*25)
for member in members:
    print(member)

firstmem = members[0]
file_content, ste, rc = execute_command(f'dcat "{pds}({firstmem})"')
print(f'{pds}({firstmem})')
print('-'*25)
print(file_content)

```

While similar to the previous example, there are some differences:

- The `tsocmd "listds \'{pds}\' members"` command used in the previous script, retrieves an output including headers and information that has to be parsed before finding the list of members. That's why we needed the following sub-list to start in the output record #6 (Python indexes starting in 0) `members = members.split("--MEMBERS--")[1]`. The command output is as follows:

```

PROD001.TENNIS
--RECFM=LRECL-BLKSIZE=DSORG
  FB      80      27920      PO
--VOLUMES--
  USER01
--MEMBERS--
  FRANCE
  SERBIA
  USA

```

- The ZOAU `f'mls "{pds}"` vs `f'tsocmd "listds \'{pds}\' members'"` command doesn't require the escape characters (`\'`) and avoids the need of playing with the double quotes and single ones making the code much more human readable.
- The `tsocmd` addresses TSO in MVS and we can set the dataset name as it is. The `cat` command needs slashes as a prefix to our dataset to identify this is a file in the MVS space `f'cat "//\'{pds}({firstmem})\''`. The command we are using from ZOAU doesn't need slashes: `f'dcat "{pds}({firstmem})"`.

💡 Tip: When migrating JCLs to ZOAU:

To illustrate ZOAU usability, here's a common JCL to print a file:

```
//PRINTMEM JOB (124400000), 'MYDEPT', CLASS=A, MSGCLASS=X,  
//      NOTIFY=&SYSUID  
//DCAT EXEC PGM=IEBGENER  
//SYSPRINT DD SYSOUT=*  
//SYSIN DD DUMMY  
//SYSUT1 DD DSN=PROD001.TENNIS(FRANCE), DISP=SHR  
//SYSUT2 DD SYSOUT=*
```

The ZOAU `mvscmd` executes any z/OS program and DDs are passed as `--ddname`. Therefore a JCL can be coded in a similar way:

```
mvscmd --pgm=IEBGENER --sysprint=* --sysin=dummy --sysut2=* --  
sysut1="PROD001.TENNIS(FRANCE)"
```

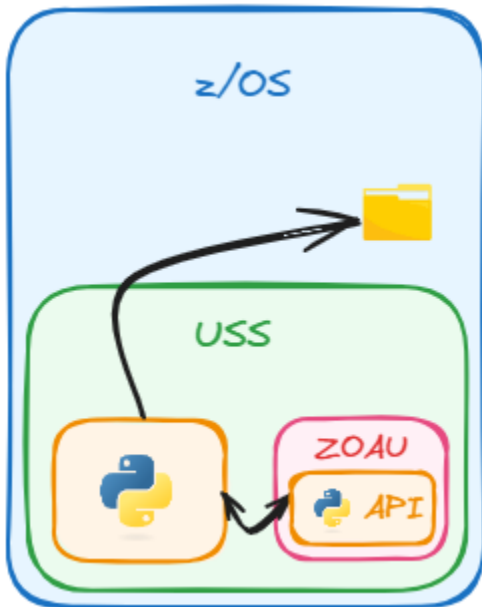
Since this functionality is already included within ZOAU command, this simplifies the code using it to print the dataset:

```
dcat "PROD001.TENNIS(FRANCE)"
```

For more complex structures, `mvscmd` manages DDs the same way a JCL does. Here are basic examples:

```
--ddname=* -> standard output  
--ddname=dummy -> sysin as dummy  
--ddname=PROD001.TENNIS, EXCL -> DISP=OLD  
--ddname=PROD001.TENNIS.COPY, NEW, TYPE=pdse, LRECL=80, BLKSIZE=32720  
--ddname=/u/users/zwe/zowe.yaml, mod -> UNIX file allocated as MOD
```


Python & ZOAU APIs



Embedding ZOAU in scripts leverages management and interaction with z/OS resources, but using Python has an advantage: The ZOAU Python API. ZOAU provides easy-to-use class libraries for accessing MVS resources, without requiring installation or configuration of other software.

The Python APIs include several classes, functions and modules, including: `datasets`, `gdgs`, `jobs`, `mvscdm`, `opercmd`, `zoau_io`, `system`, and `ztype`.

There is an additional class in ZOAU, the `'exceptions'`. A Python exception is an error detected during execution handled and managed with the code. Because the ZOAU Python API extends Python's Exception class, a ZOAU exception is handled in the same way as a built-in exception.

For example, we can raise an exception if trying to write a dataset and there is a problem by using the `'DatasetWriteException'` exception. We can then access to the different attributes of the object that has been generated, like the `rc`, the `Stdout` or the `Stderr`:

```
from zoutil_py import datasets
from zoutil_py.exceptions import DatasetWriteException

spain = ''' Rafael Nadal
Carlos Alcaraz
David Ferrer
Juan Carlos Ferrero
Fernando Verdasco
'''

try:
```

```

    datasets.write("PROD001.TENNIS(SPAIN)", content=spain)
except DatasetWriteException as write_exception:
    if write_exception.response.rc == 99:
        check_file()

```

The next script won't be using `subprocess.run` to execute a command. Instead we will call the ZOAU Python APIs `'datasets.list_members'` and `'datasets.read'`.

For this use case, import the `'datasets'` function from `'zoautil_py'`. We can also import other functions included in ZOAU to call MVS programs, command execution, Data sets and JES utilities.

```

# Python script in USS using ZOAU Python APIs-----
# - Retrieve all member names from a PDS: PROD001.TENNIS
# - Retrieve the content of the first member
#-----
from zoautil_py import datasets

pds = 'PROD001.TENNIS'

members = datasets.list_members(f'{pds}')
print(f'{pds}')
print('-'*25)
for member in members:
    print(member)

frstmem = members[0]
file_content = datasets.read(f'{pds}({frstmem})')
print(f'{pds}({frstmem})')
print('-'*25)
print(file_content)

```

The code has been **reduced to 5 lines** without the print statements.

Now let's display the content of the first member, through another method. This creates an `IEBGENER` JCL dynamically in code, resolving the variables for the pds and first member in the `SYSUT1 DD`.

Then run the following steps:

1. Save the data in the pds with the name `'JCL'` by calling the ZOAU Python API: `'datasets.write'`
2. Submit with `'jobs.submit'` and retrieve the jobid. The Job class can retrieve many different parameters from the job execution, such as the id, name, owner, status, rc, ...
3. Delete the JCL member: `'datasets.delete_members'`
4. And finally use the `'jobs.read_output'` to get the content of the first member into the `'file_content'` variable.

```

# Python script in USS using ZOAU Python APIs and JCL -----
# - Retrieve all member names from a PDS: PROD001.TENNIS
# - Retrieve the content of the first member via JCL
#-----
from zooutil_py import datasets, jobs

pds = 'PROD001.TENNIS'

members = datasets.list_members(f'{pds}')
print(f'{pds}')
print('-'*25)
for member in members:
    print(member)

frstmemb = members[0]

JCL = f'''//PROD001X JOB (124400000), 'MYCOMP', CLASS=A, MSGCLASS=X,
//      NOTIFY=&SYSUID
//STEP01 EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=*
//SYSUT1 DD DSN={pds}({members[0]}), DISP=SHR
//SYSUT2 DD SYSOUT=*
//SYSIN DD DUMMY'''

datasets.write(f'{pds}(JCL)', content=JCL)

jobid = jobs.submit(f'{pds}(JCL)').id

datasets.delete_members(f'{pds}(JCL)')

file_content = jobs.read_output(f'{jobid}', 'STEP01', 'SYSUT2')

print(f'{pds}({frstmemb})')
print('-'*25)
print(file_content)

```

💡 Tip: The ZOAU option without the Python API would use the following commands and the 'subprocess.run'

1. Write a dataset or member: 'decho'
2. Submit a JCL: 'jsub'
3. Delete a member: 'mrm'
4. List job output: 'pjdd'

Remote connection

The examples, so far, have Python executing on USS. The scripts have performed actions on MVS but they can also be executed off host, such as distributed pipeline or another z/OS system. The same script can be run against multiple LPARs, such as maintenance tasks, with little to no modifications.

SSH or z/OSMF RESTful APIs can also be used to perform the actions remotely.

SSH

SSH (Secure Shell) is a network protocol used to securely access and manage remote systems over an encrypted connection. It ensures confidentiality and integrity of data during communication, commonly used for secure login, command execution, and file transfers.

Authentication Methods

There are four common authentication methods:

1. **Username and Password:** Simple, but vulnerable to attacks.
2. **Public/Private Key Authentication:** Highly secure, but computationally expensive.
3. **Ticket-Based Authentication:** Improved security, but relies on server management.
4. **Digital Certificates:** Verifies user and device identity, but complex and expensive.

Factors to Consider

- **Complexity:** Public/private key and digital certificates are more complex.
- **Security Level:** These methods offer higher security, but at a cost.
- **Compliance:** Some organizations may require specific methods for regulations.

The following examples use username and password authentication. While easy to test, this is not recommended for production or automation.

Credentials should never be stored with code. To address this, configuration files can be used to keep the credentials separate from code. The configuration file will use key/value pairs to store the information. Set the following values for the remote connection:

- It is common practice to use a configuration file in YAML or JSON format since Python (and most scripting languages) have methods to deal with them both, though any format can be used. The `'config.yaml'` file will look like this:

```
host: my.host.url
username: myUserId
password: myPasswd
```

- This Python code reads the config file at the beginning of the script. The values are accessed using the name of the variable, confile, and the key (['host']):

```
with open('config.yaml', 'r') as f:
    confile = yaml.safe_load(f)

my_host_is = confile['host']
```

- To use the YAML feature we need to import the YAML library to make it available for our code:

```
'import yaml'
```

- The script also needs the 'Paramiko' library. Paramiko is a Python library for SSH (Secure Shell) and SFTP (Secure File Transfer Protocol). It enables secure connections to remote machines. This enables command execution, file transfers, and task automation.

Executing Python scripts from external systems, like a Windows machine, requires careful handling of environment variables and shell behavior to ensure proper integration with the mainframe environment.

When logging into z/OS UNIX System Services (USS) via the command line, the shell automatically loads environment configurations such as 'PYTHONPATH', 'ZOAU_HOME', and other critical variables. These configurations come from initialization files like '.profile' or '.bashrc'.

However, when running scripts via SSH, these environment variables are not automatically loaded unless explicitly handled. This discrepancy can result in errors like 'ModuleNotFoundError' when importing ZOAU modules such as 'zoautil_py'.

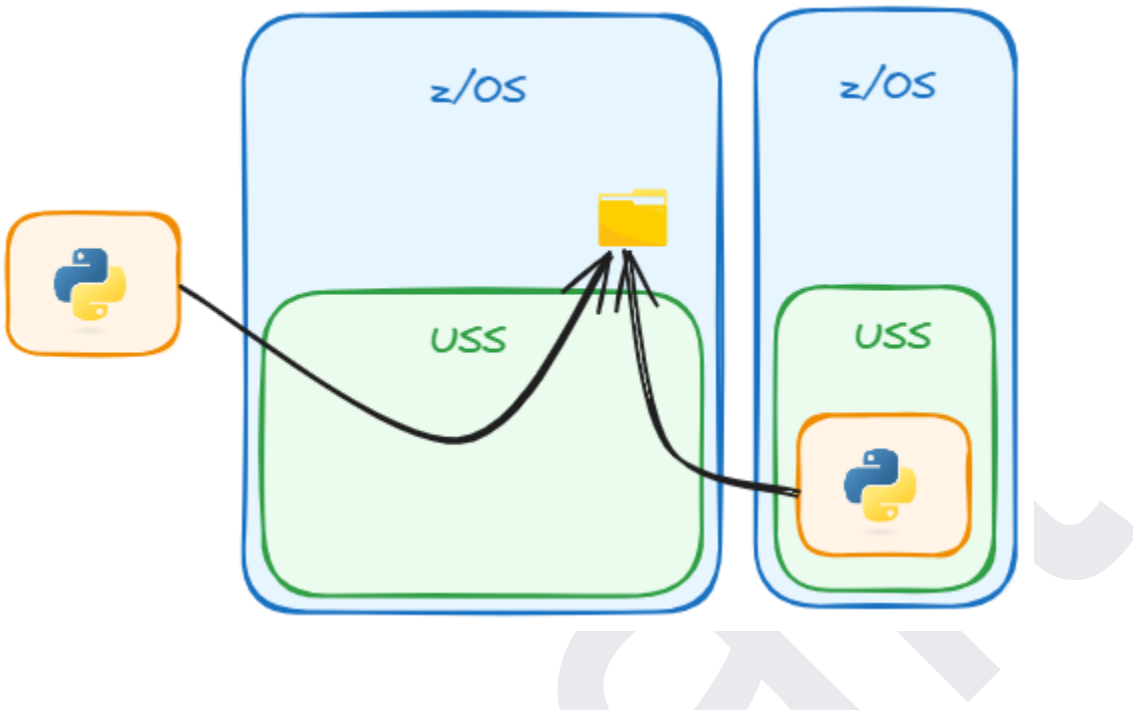
To address this, two main methods are available:

- Explicitly providing environment data in the SSH command
- Using a login shell with the 'sh -L' method

Here are 3 samples:

- Python & SSH: This is the first use case with Python and USS where shell commands are executed. Since no ZOAU modules will be used, there's no need to define the location.
 - Python & SSH & ZOAU Explicit Environment: Using ZOAU, the environment variables must be explicitly defined.
 - Python & SSH & ZOAU API Login shell: The ZOAU Python API with the login shell method is used.
-

Python & SSH



```
# Python script SSH Basic USS -----  
# - Retrieve all member names from a PDS: PROD001.TENNIS  
# - Retrieve the content of the first member  
#-----  
import yaml  
import paramiko  
  
with open('config.yaml', 'r') as f:  
    confile = yaml.safe_load(f)  
  
host = confile['host']  
username = confile['username']  
password = confile['password']  
  
# Initialize the SSH client  
client = paramiko.SSHClient()  
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())  
  
def execute_command(command):  
    client.connect(host, port=22, username=username, password=password)  
    # Execute the remote command  
    stdin, stdout, stderr = client.exec_command(command)  
    out = stdout.read().decode()  
    client.close()
```

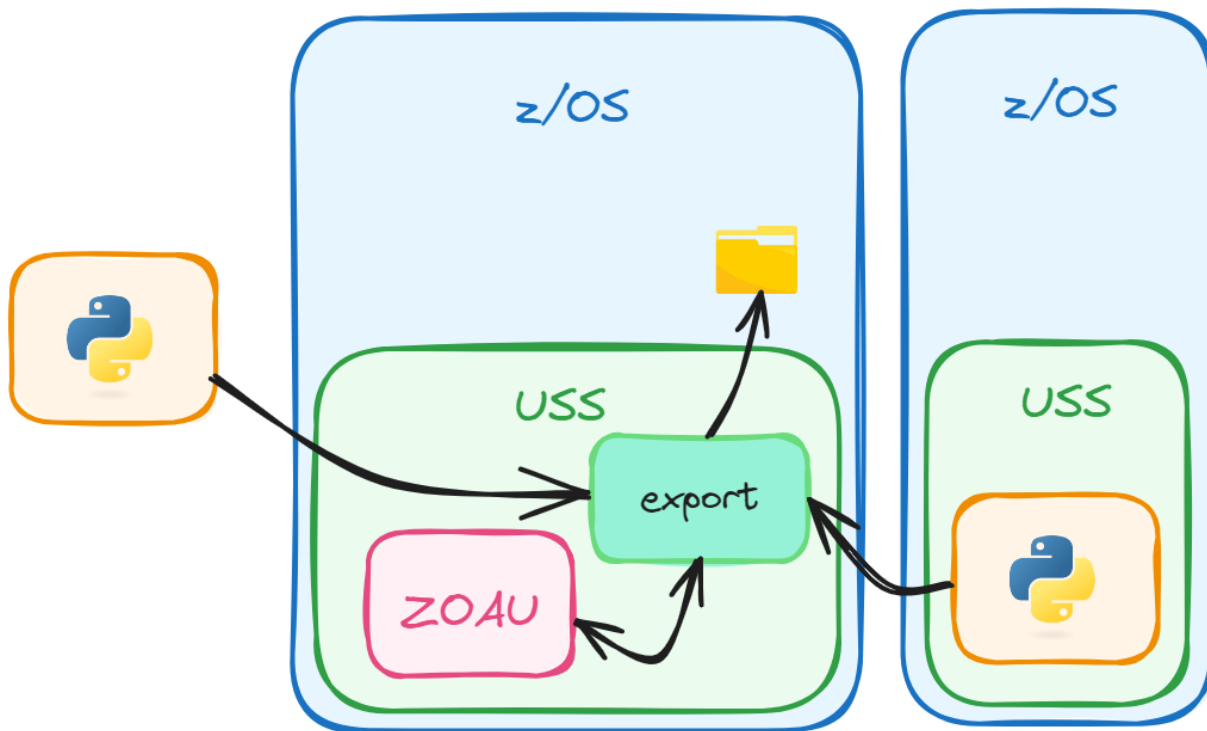
```
    return out

pds = 'PROD001.TENNIS'

members, ste, rc = execute_command(f'tsocmd "listds \'{pds}\'  
members"')
members = [x.strip() for x in members.split('\n')]
members = members.split('--MEMBERS--')[0]
print(f'{pds}')
print('-'*25)
for member in members:
    print(member)

frstmem = members[0]
file_content = execute_command(f'cat "/*\'{pds}({frstmem})\'"'')
print(f'{pds}({frstmem})')
print('-'*25)
print(file_content)
```

Python & SSH & ZOAU Explicit Environment



Explicitly Providing Environment Data

This approach involves specifying all necessary environment variables directly within the SSH command. It ensures that the Python script has access to the ZOAU environment, regardless of the shell's default behavior.

```
# Python script in USS using ZOAU & SSH Explicit environment data -----
# - Retrieve all member names from a PDS: PROD001.TENNIS
# - Retrieve the content of the first member
#-----

import yaml
import paramiko

with open('config.yaml', 'r') as f:
    confile = yaml.safe_load(f)

host      = confile['host']
username  = confile['username']
password  = confile['password']

# Initialize the SSH client
```



```

client = paramiko.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

def execute_command(command):
    client.connect(host, port=22, username=username, password=password)

    exports= (
        "export
PATH=/usr/lpp/IBM/zoutil/env/bin:/usr/lpp/IBM/zoutil/bin:/usr/lpp/IBM
/cyp/v3r9/pyz/bin:/bin:/usr/lpp/java/current/bin:/usr/lpp/java/current"
        "export LIBPATH=/usr/lpp/IBM/zoutil/lib;"
    )

    remote_command = (exports + command)

    # Execute the remote command
    stdin, stdout, stderr = client.exec_command(remote_command)

    out = stdout.read().decode()
    client.close()
    return out

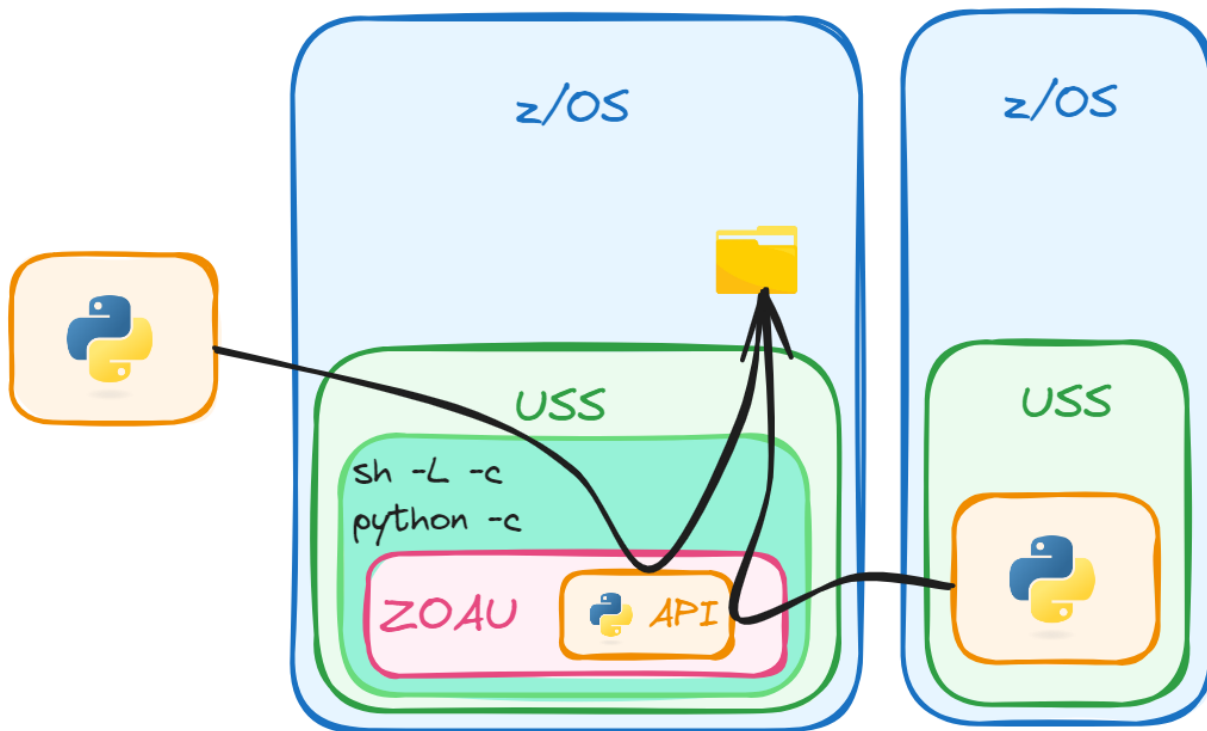
pds = 'PROD001.TENNIS'

command = f'mls "{pds}"'
members = execute_command(command)
members = [x.strip() for x in members.split('\n')]
print(f'{pds}')
print('-'*25)
for member in members:
    print(member)

frstmem = members[0]
file_content = execute_command(f'dcat "{pds}({frstmem})"')
print(f'{pds}({frstmem})')
print('-'*25)
print(file_content)

```

Python & SSH & ZOAU API Login shell



Using a Login Shell with ' sh -L '

An alternative is to invoke a login shell (' sh -L '), which automatically loads the environment variables from the user's initialization files (e.g., '.profile').

When using the ZOAU Python API, the library ('zoautil_py') must be imported into the script. When executing the script from off host, the library location must be explicitly defined.

A couple of comments on this case:

- Be very careful with apostrophes, double apostrophes and escape characters. This is especially important when nesting different quote levels.
- In this case, the shell is executing the Python script, then returning.

Here is a sample of how to achieve this:

```
# Python script in USS using ZOAU Python APIs & SSH Login Shell -----  
# - Retrieve all member names from a PDS: PROD001.TENNIS  
# - Retrieve the content of the first member  
#-----  
import yaml  
import paramiko
```

```

with open('config.yaml', 'r') as f:
    confile = yaml.safe_load(f)

host      = confile['host']
username  = confile['username']
password  = confile['password']

# Initialize the SSH client
client = paramiko.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

def execute_command(command):
    client.connect(host, port=22, username=username, password=password)

    login = (
        f"sh -L -c 'python -c \"from zoutil_py import datasets; \""
    )

    remote_command = (login + command)

    # Execute the remote command
    stdin, stdout, stderr = client.exec_command(remote_command)

    out = stdout.read().decode()
    client.close()
    return out

pds = 'PROD001.TENNIS'

command = f"print(datasets.list_members(\\\\"{pds}\\\"))\\\""

members = execute_command(command)
# Convert the output to a list since it is newline-separated
members = members.splitlines()
print(f'{pds}')
print('-'*25)
# Convert the string to a list
members = members[-1].strip("[']").replace("'", "").split(", ")
for item in members:
    print(item)

frstmem = members[0]
command = f"print(datasets.read(\\\\"{pds}\\\"({frstmem})\\\"))\\\""
file_content = execute_command(command)

```

```
# To get rid of the headers I know in my text there is no '=' signs
clean_output = "\n".join(
    line for line in file_content.splitlines()
    if "=" not in line
    and not line.strip().startswith("export")
)
print(f'{pds}({frstmem})')
print('-'*25)
print(clean_output)
```

z/OSMF

Addressing remote servers can be also done via z/OSMF RESTful APIs.

z/OSMF (z/OS Management Facility) is a web-based interface for managing IBM z/OS systems. It simplifies tasks such as configuration, monitoring, and automation through a modern, user-friendly interface, reducing reliance on traditional command-line tools.

One way to know the available REST services provided by z/OSMF is accessing the Swagger interface to display the APIs information. In order to use the APIs, access must be granted by the security administrator.

The Swagger interface can be used to explore, test and run z/OSMF REST APIs without the need to write the program and is the best source for the APIs documentation. It can be accessed using the following URL, substituting the appropriate items: `https://<hostname>:<port>/zosmf/api/explorer`

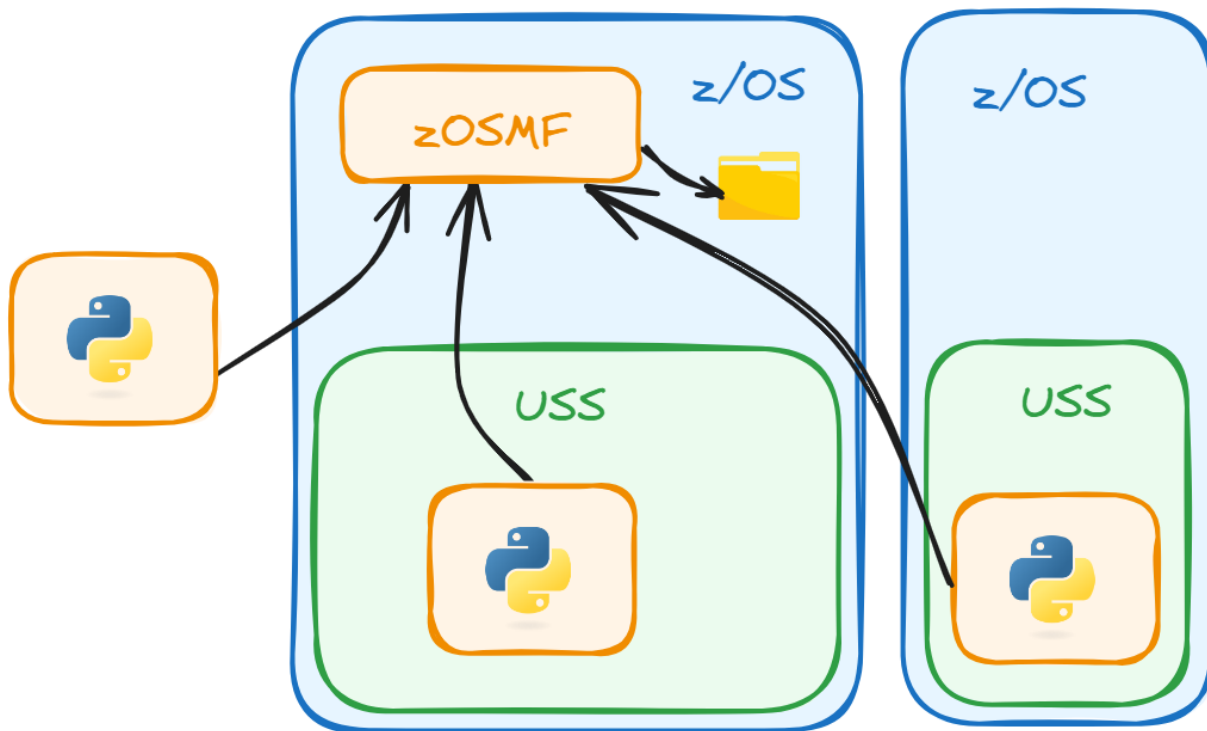
The script will use the 'requests' library.

The requests Python library is a user-friendly HTTP library used to send HTTP/1.1 requests, such as GET, POST, PUT, DELETE, etc., to interact with web services and APIs. It simplifies making web requests and handling responses by providing features like:

- Sending parameters, headers, and data with requests.
- Handling cookies and sessions.
- Automatic decoding of JSON responses.
- Managing timeouts and error handling.
- Support for HTTPS and authentication mechanisms.

It's widely used for making REST API calls and web scraping due to its simplicity and powerful features.

Python & z/OSMF



This use case will use two services:

- `/zosmf/restfiles/ds/{dataset-name}/member` to list all members of the dataset
- `/zosmf/restfiles/ds/{dataset-name}({member-name})` to retrieve the data from a member

IBM has provided a lot of optional parameters in the services. The parameters can be used to limit the data return, saving a lot of coding.

The sample scripts will read a local configuration file containing credentials. In this case credentials are user:password encoded in Base64 format, urls, etc. stored in a YAML format file with standard {key:value} pairs. The default z/OSMF port is 443, but it may be different for each LPAR:

```
base_uri: https://my.company.mainframe.net
api_port: 443
credentials: dXNlcmlkOnBhc3N3b3Jk 😊
```

Within the code, a variable named `'base_path'` defines the desired service, in this case: `'zosmf/restfiles'`.

```
# Python script z/OSMF -----
# - Retrieve all member names from a PDS: PROD001.TENNIS
# - Retrieve the content of the first member
#-----
```

```

import yaml
import requests

with open('config.yaml', 'r') as f:
    confile = yaml.safe_load(f)

base_uri = confile['base_uri']
api_port = confile['api_port']
credentials = confile['credentials']

def get_data(type=''):
    url = f'{base_uri}:{api_port}/{base_path}/{parms}'
    headers = {
        'Authorization': f'Basic {credentials}',
        'Content-Type': 'application/json',
        'X-CSRF-ZOSMF-HEADER': ''
    }
    response = requests.get(url, headers=headers, verify=True)
    if type == 'text': return response.text
    else: return response.json()

pds = 'PROD001.TENNIS'
base_path = 'zosmf/restfiles'

# list members
parms = f'ds/{pds}/member'
members = get_data()
print(f'{pds}')
print('-'*25)
members = [item['member'] for item in members['items']]
for member in members:
    print(member)

# Retrieve the content for the first member [0]
member = members[0]
parms = f'ds/{pds}({member})'
file_content = get_data('text')
print(f'{pds}({member})')
print('-'*25)
print(file_content)

```

Zowe

Zowe is an open-source framework that provides modern tools and APIs for interacting with IBM z/OS mainframes. It bridges traditional mainframe systems with modern DevOps practices, enabling developers, system administrators, and operators to work more efficiently. Zowe includes three main components:

- Zowe CLI: A command-line interface to automate mainframe tasks and integrate them into modern development pipelines.
- Zowe API Mediation Layer: A gateway for securely accessing and managing z/OS resources through REST APIs.
- Zowe Desktop: A web-based interface for managing mainframe resources through a user-friendly GUI.

It simplifies mainframe access, enhances developer productivity, and promotes interoperability across tools and platforms.

Zowe API ML

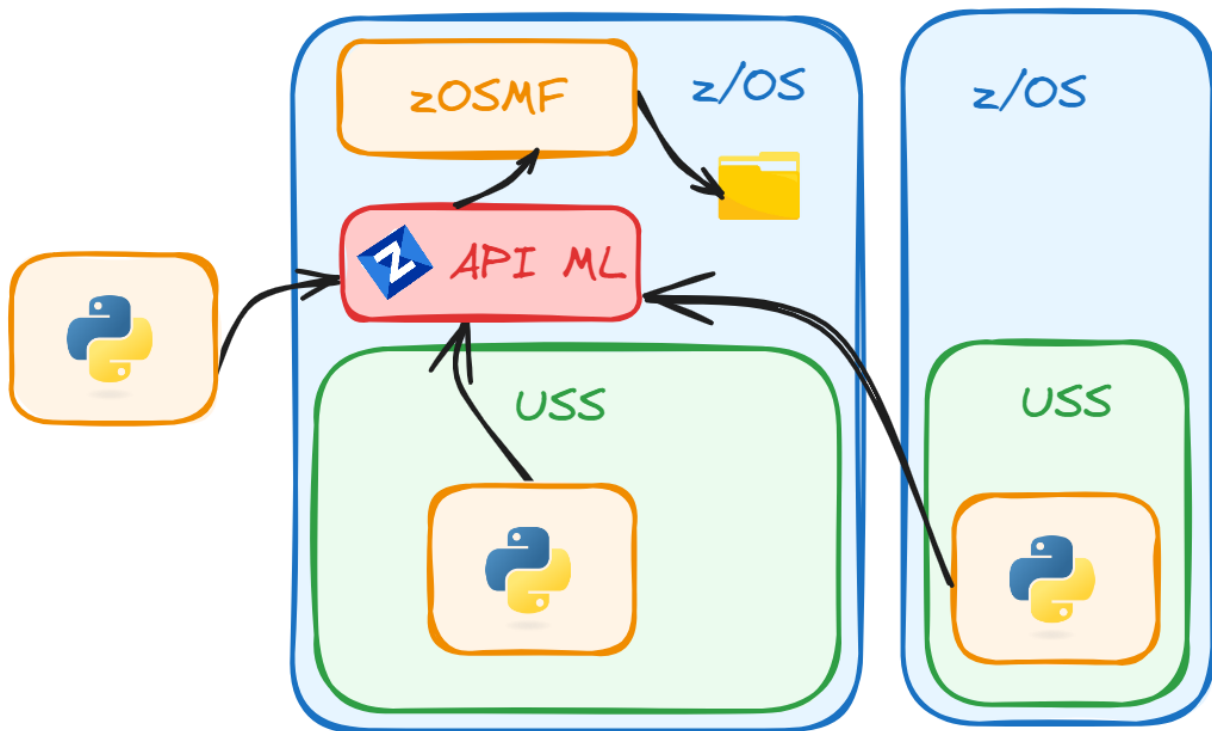
The Zowe API Mediation Layer (API ML) is a key component of the Zowe framework that acts as a gateway for managing and accessing REST APIs on IBM z/OS systems. It simplifies integration, improves security, and provides a unified interface for z/OS services.

Main Features:

- API Discovery: Centralized catalog for finding and accessing available APIs on the mainframe.
- Secure Communication: Provides authentication and authorization mechanisms (e.g., support for single sign-on and TLS).
- High Availability: Ensures reliable access to APIs through load balancing and failover support.
- Routing and Integration: Simplifies API consumption by routing requests and enabling seamless integration with other systems.
- User Interface: Offers a web-based interface to browse and explore available APIs.
- Auditing and Logging: Tracks and logs API usage for monitoring and troubleshooting purposes.

The API ML helps modernize mainframe development by enabling easier interaction with z/OS services and integrating them into enterprise-wide DevOps workflows.

Python & Zowe API ML



To access catalogued API services we just need some minor changes to our previous Python script:

Edit our '`config.yaml`' file and update it with the API ML `base_uri` and the API ML port (usually 7554).

In our code, where we have defined the `base_path` to address the restfiles service, we need to change from: '`zosmf/restfiles`' to '`ibmzosmf/api/v1/zosmf/restfiles`'

Using API ML allows us to access all z/OSMF API services in addition to any other conformant API discovered & catalogued products or applications through a unique point of access with 2 main benefits:

- Enhance the user experience letting API ML address our request to the correct service without the need of updating ports and other configuration options.
- More secured policy since there is only one port, acting as a gateway, exposed for all available APIs.

```
# Python script & Zowe API ML-----  
# - Retrieve all member names from a PDS: PROD001.TENNIS  
# - Retrieve the content of the first member  
#-----  
import yaml
```

```

import requests

with open('config.yaml', 'r') as f:
    confile = yaml.safe_load(f)

base_uri = confile['base_uri']
api_port = confile['api_port']
credentials = confile['credentials']

def get_data(type=''):
    url = f'{base_uri}:{api_port}/{base_path}/{parms}'
    headers = {
        'Authorization': f'Basic {credentials}',
        'Content-Type': 'application/json',
        'X-CSRF-ZOSMF-HEADER': ''
    }
    response = requests.get(url, headers=headers, verify=True)
    if type == 'text': return response.text
    else: return response.json()

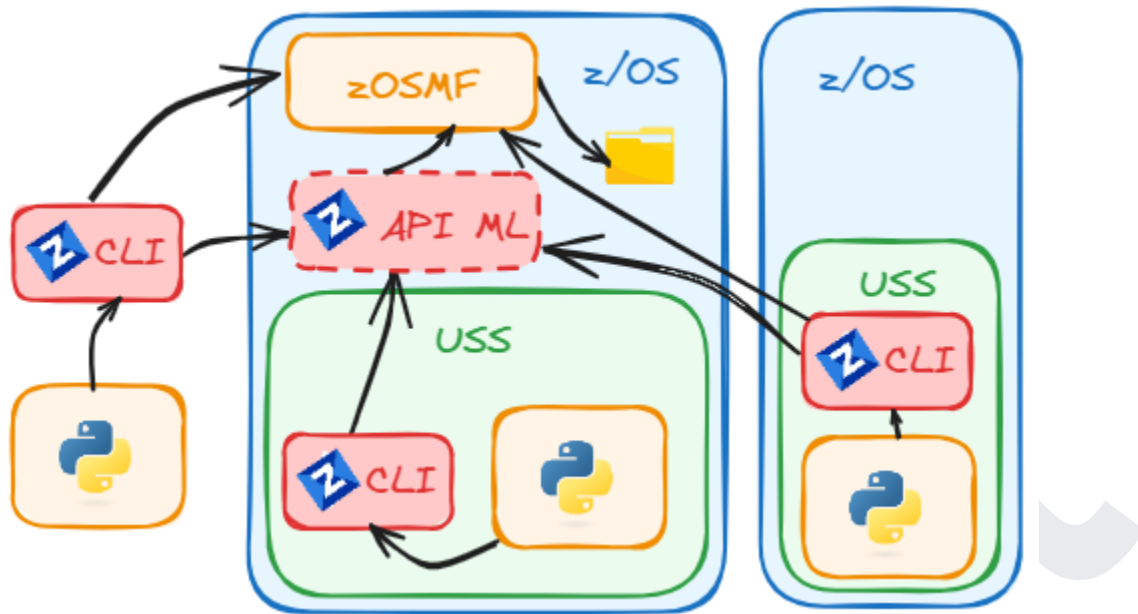
pds = 'PROD001.TENNIS'
base_path = 'ibmzosmf/api/v1/zosmf/restfiles'

# list members
parms = f'ds/{pds}/member'
members = get_data()
print(f'{pds}')
print('-'*25)
members = [item['member'] for item in members['items']]
for member in members:
    print(member)

# Retrieve the content for the first member [0]
member = members[0]
parms = f'ds/{pds}({member})'
file_content = get_data('text')
print(f'{pds}({member})')
print('-'*25)
print(file_content)

```

Python & Zowe CLI



Zowe CLI allows us to power our Python scripts by embedding commands that can be run from a command prompt directly on the code.

There are different ways to authenticate the user running the CLI commands against a Mainframe instance. In Zowe v1, profiles were used to define the connection properties. From Zowe v2 we can use a Team Configuration file, where settings are easier to manage and configure and can be shared among different users.

The API ML supports tokens for authentication services, including API ML SSO (Single Sign On).

Zowe V1 Profiles have been fully deprecated. Using Zowe V3, it uses Team Configuration files. As such, the following use case assumes the configuration file is configured properly and in use.

The `'execute_command'` function will be used again:

```
# Python script & Zowe CLI -----
# - Retrieve all member names from a PDS: PROD001.TENNIS
# - Retrieve the content of the first member
#-----
import subprocess

def execute_command(command):
    try:
        result = subprocess.run(command,
                                shell=True, text=True, capture_output=True)
```

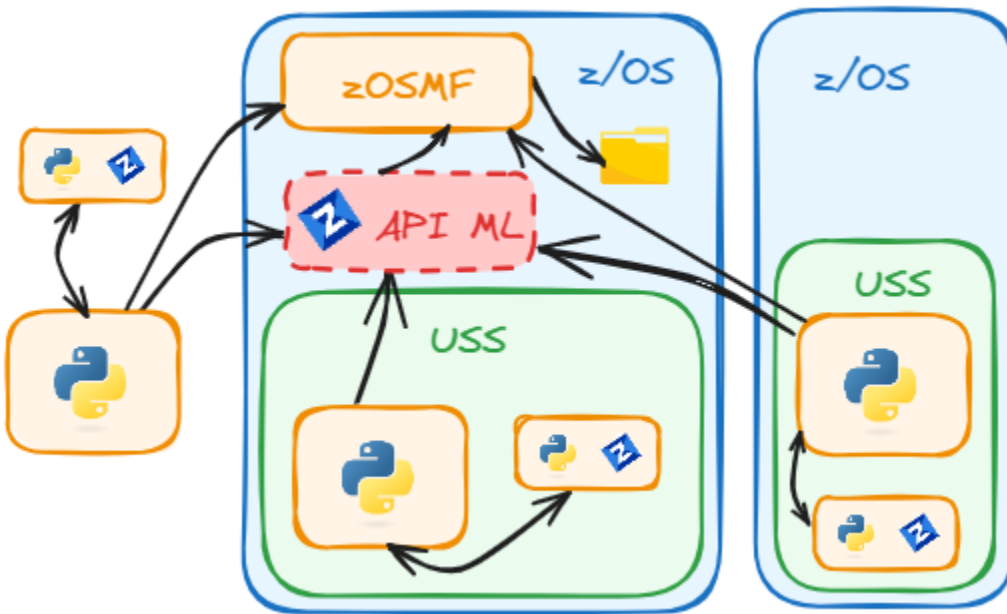
```
except subprocess.CalledProcessError as e:
    print("Command execution failed.")
    print("Return Code:", e.returncode)
    print("Error:", e.stderr)
    return result.stdout, result.stderr, result.returncode

pds = 'PROD001.TENNIS'

command = f'zowe zos-files list all-members "{pds}"'
members, ste, rc = execute_command(command)
print(f'{pds}')
print('-'*25)
print(members)

members = members.splitlines()
frstmem = members[0]
command = f'zowe zos-files view data-set "{pds}({frstmem})"'
file_content, ste, rc = execute_command(command)
print(file_content)
```

Python & zowe SDK



The Zowe Client Python SDK, is a set of Python packages that allows us to interact with z/OS REST API interfaces the same way we do with Zowe CLI but without the need of Node.js implemented as a prerequisite.

For credentials management we can use our Team Configuration profile or directly provide the configuration parameters stored in a file and make it accessible for our code in a local dictionary:

```
# Python script & Zowe SDK -----
# - Retrieve all member names from a PDS: PROD001.TENNIS
# - Retrieve the content of the first member
#-----
import yaml
from zowe.zos_files_for_zowe_sdk import Files

with open('config.yaml', 'r') as f:
    confile = yaml.safe_load(f)

profile = {
    "host": f'{confile["host"]}',
    "port": f'{confile["api_port"]}',
    "user": f'{confile["username"]}',
    "password": f'{confile["password"]}'
}
```

```
dsn = 'PROD001.TENNIS'
files = Files(profile)

members = files.list_dsn_members(f'{dsn}')
members = [item['member'] for item in members['items']]
for member in members:
    print(member)

frstmem = members[0]
file_content = files.get_dsn_content(f'{dsn}({frstmem})')
print(file_content)
```

Samples

The use cases, so far, have been focused on a simple task. Listing and viewing a dataset is pretty basic. There are far more interesting use cases to be explored.

When creating automation, it must be defined and scoped. From there, it can be optimized.

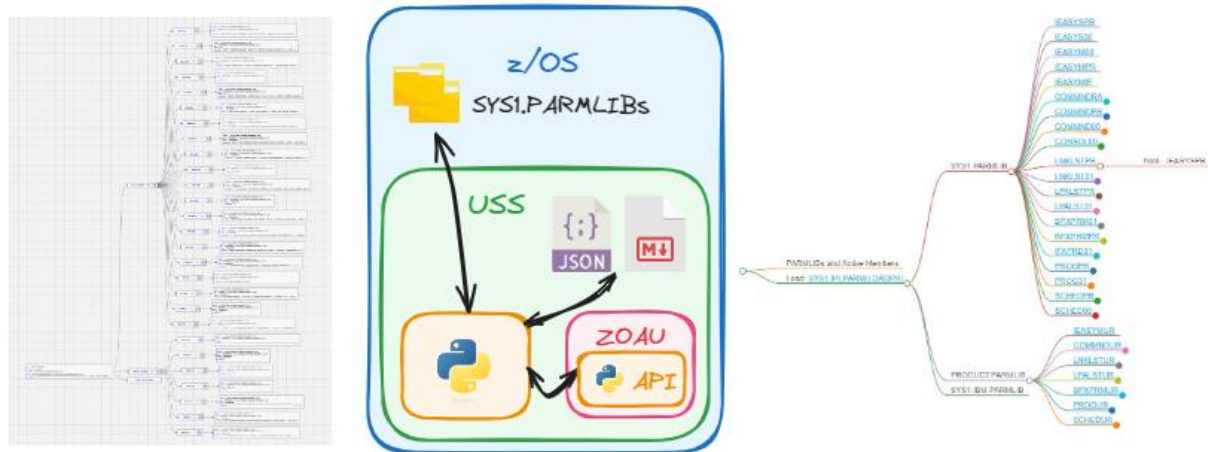
Automation requires components to be available where they will be executed. This can be on the automation platform itself, or be available to the location where it will be executed. For example, if using Jenkins, the code could reside on the same server, in a git repository, or on the platform where it will execute. The scripts must be able to execute against the target. This could be using Windows to call mainframe web services or actually execute on the mainframe itself. In addition to the executable, such as Zowe, any libraries used would also need to be available on the target platform.

Security is extremely important. Regardless of the access method (Zowe CLI, Web Services API, APL Mediation Layer), credentials and passwords should be secured. Tools like Jenkins have a credential store. This allows users to execute commands without having direct access to the credentials. Regardless of the method chosen, security should be the first concern for any automation.

Output can be formatted in different formats. This can include XML, JSON, straight text, tables, comma separated values, etc. This is important as it makes it easier to manipulate or use the data in various systems.

This use case is generic, but intended to illustrate a way to use the data. The goal is to provide ideas and show automation. These settings aren't exhaustive but an illustration on tackling a problem.

IPL Map USS ZOAU Python APIs



USE CASE: IPL Mapping

A system programmer often deals with multiple LPARs with different settings. They often need to check or update settings, so they have to identify the IPL map to find active PARMLIBs and member concatenations and where they are referenced.

What if there was an automated method for gathering this information and displaying it graphically, so it is easier to understand? And what if it included information about the startup chain, as well?

VS Code has the ability to display information in multiple formats. Could this information be collected, correlated and displayed in a way making it easier to understand? VS Code can do that.

To do this, a Python dictionary can be used. It would contain the IPL concatenation for each PARMLIB and store relevant information, such as the members requested by the script.

path: path to local folder and file to be used as a link
md: the same but suited to be on a markdown document
from: IEASYS member that points to this member
content: Content of the member for directory browse

The dictionary contains information about the SYS1.IPLPARM(LOADxx) member at the beginning.

This script will run from USS using ZOAU Python APIs. It will retrieve most of the information. ZOAU provides commands to gather information from the IPL chain and access to z/OS files and resources.

Credentials, in this use case, are managed through the USS connection. This particular scenario runs on z/OS 3.1, Python 3.11, and ZOAU 1.2

Here's the representation of the IPL data.


```

{
  "ds": "SYS1.IPLPARM",
  "loadxx": "LOADPR",
  "path": "./data/SYS1.IPLPARM(LOADPR).txt",
  "md": "[SYS1.IPLPARM(LOADPR)](./data/SYS1.IPLPARM(LOADPR).txt)",
  "content": <file-content>,
  "SYS1.PARMLIB": {
    "IEASYSR": {
      "path": "./data/SYS1.PARMLIB(IEASYSR).txt",
      "md": "[IEASYSR](./data/SYS1.PARMLIB(IEASYSR).txt)",
      "from": "",
      "content": <file-content>
    },
    "COMMND00": {
      "path": "./data/SYS1.PARMLIB(COMMND00).txt",
      "md": "[COMMND00](./data/SYS1.PARMLIB(COMMND00).txt)",
      "from": "IEASYSR",
      "content": <file-content>
    },
    ...
    "PRODUCT.PARMLIB": {
      "IEASYMUR": {
        "path": "./data/PRODUCT.PARMLIB(IEASYMUR).txt",
        "md": "[IEASYMUR](./data/PRODUCT.PARMLIB(IEASYMUR).txt)",
        "from": "",
        "content": <file-content>
      },
      "COMMNDUR": {
        "path": "./data/PRODUCT.PARMLIB(COMMNDUR).txt",
        "md": "[COMMNDUR](./data/PRODUCT.PARMLIB(COMMNDUR).txt)",
        "from": "IEASYSR",
        "content": <file-content>
      },
      ...
    }
  }
}

```

Like any other software, different versions of software support different features. This includes versions of Python, ZOAU, various libraries, even zOS itself. If a script doesn't work, check the versions of code on the system being used. Logic can be applied to address changes when limits or workarounds are needed.

Here's an explanation of each code block, the full code is at the end.

The Python API libraries from ZOAU must be loaded into the script:

```

from zoutil_py import opercmd
from zoutil_py import datasets

```

```
from zoutil_py import zsystem
from zoutil_py import jobs
```

This imports standard libraries to handle JSON and operating system files:

```
import json
import os
```

Create a folder to store the content of the members:

```
path = f'./data'
if os.path.exists(path) == False:
    os.mkdir(path)
```

To get the IPLPARM DSN, the LOADxx member, the IEASYS and IEASYM concatenation the 'd iplparm' command is issued. This can be used for console commands.

```
# Execute ZOAU oper command
def oper_command(command, parameters, terse=True):
    out = opercmd.execute(command=f"{command}",
        parameters=f"{parameters}", terse=True)
    if out.rc == 0:
        out = out.stdout_response
    else:
        print(out.stdout_stderr_response)
        exit(1)
    return out

# Get load_param_dsn, loadxx, ieasym_list, ieasys_list
out = oper_command('d', 'iplinfo')
```

The `terse=True` parameter provides terse output, only the command response is printed to the terminal. The `opercmd -t "d iplinfo"` command returns information. This information should be parsed to retrieve the LOADXX suffix and the values for IEASYM and IEASYS. The last 'L' in IEASYM and 'OP' in IEASYS are not needed:

```
# opercmd -t "d iplinfo"
IEE254I 12.46.38 IPLINFO DISPLAY 295
SYSTEM IPLED AT 13.24.40 ON 12/28/2024
RELEASE z/OS 03.01.00 LICENSE = z/OS
USED LOADPR IN SYS1.IPLPARM ON 04006
ARCHLVL = 2 MTLSHARE = N
VALIDATED BOOT: NO
IEASYM LIST = (00,PR,IP,UR,L)
```

```
IEASYS LIST = PR (OP)
IODF DEVICE: ...
```

The Python code parses this information. It also adds '00' to the IEASYS list since it always exists. The code ends up like:

```
IEASYM = ['00', 'PR', 'IP', 'UR']
IEASYS = ['PR', '00']
```

This is loaded into the dictionary, 'iplmap'.

There are several ways to get PARMLIB concatenation. SYS1.IPLPARM can be parsed, but the ZOAU command, pparm, returns the exact information. The ZOAU Python API command is:

```
parmlibs = zsystem.list_parmlib()
```

The ZOAU 'parmwhence <member>' command, retrieves the PARMLIB where the member is being called first. The ZOAU Python API command is:

```
parmlib = zsystem.find_parmlib(f'"{member}"')
```

The script's main function 'find_members', retrieves the roots (i.e. IEASYM), a list of suffixes and applies them to the dictionary as key/value pairs under the corresponding PARMLIB.

For IEASYM and IEASYS the dict 'from' field will be empty since they are loaded from LOADxx. The other IEASYS concatenations must be identified. The IEASYS members could be parsed, but then the System Symbols would need to be resolved.

SDSF has an option panel that displays the System Parameters: SYSP - System Parameters.

Using the PARM, Value, Member, and the RefName fields:

SDSF SYSTEM PARAMETERS			IP01	IP01	LINE 64-74 (74)
COMMAND INPUT ==>			SCROLL ==> CSR		
NP	PARM	Value	Member	RefName	
	SCH	(PR,00,UR)	IEASYS	SPR	SCHED

This information can be accessed with a simple REXX script:

```
/* rexx */
IsfRC = isfcalls( 'ON' )
address SDSF 'ISFEXEC SYSP'
do i = 1 to isfrows
    say PARM.i VALUE.i MEMBER.i REFNAME.i
end
```

The information can be gathered using JCL. Use a temporary JCL to execute this script, submit it and retrieve the DD file that gives me the output generated by the REXX.

Manually set a list with the parameter members to manage in this sample.

```
parmlib_members = ['LNK', 'CMD', 'CON', 'LPA', 'OMVS', 'PROD', 'PROG', 'SCH']
```

When the JCL is finished the output DD from the `jobs.read_output` ZOAU Python API, provides the jobname, the stepname and the dataset needed.

Parse this DD output and retrieve only the values needed in and for each of them, then call the `'find_members'` function.

This creates a dictionary and generates a json file from the System Symbols.

Here is the whole script:

```
# Script to get the IPL map
# Run in USS
# Items generated:
#   ipl_map.json - Info of LOADxx and PARMLIBs
#   ipl_map.md - Markdown Format
#   symbols.json

from zoutil_py import opercmd
from zoutil_py import datasets
from zoutil_py import zsystem
from zoutil_py import jobs
import json
import os
import time

# Functions -----

# Execute ZOAU oper command
def oper_command(command, parameters, terse=True):
    out = opercmd.execute(command=f"{command}", parameters=f"{parameters}",
terse=True)
    if out.rc == 0:
        out = out.stdout_response
    else:
        print(out.stdout_stderr_response)
        exit(1)
    return out
```

```

# Read dataset and copies it into ./data/
#   in the format './data/dsn(member.txt)'
def dataset_read(dsn, member):
    file = f'{dsn}({member})'
    content = datasets.read(f'{file}')
    # Save the member in ./data/
    with open(f'./data/{file}.txt', 'wt') as o:
        o.write(content)
    return content

# Get PARMLIB members for a type & Suffixes
# i.e. {iplmap}, 'IEASYM', ['00', 'PR', 'IP', 'UR'], 'called from'
def find_members(iplmap, type, type_list, from_mem=''):
    for suffix in type_list:
        member = f'{type}{suffix}'
        parmlib = zsystem.find_parmlib(f'{member}')
        if parmlib == '': continue
        print(f'Processing {parmlib} {member}') # dxr
        iplmap[parmlib][member]={}
        iplmap[parmlib][member]['path'] = f'./data/{parmlib}({member}).txt'
        iplmap[parmlib][member]['md'] =
f'[{member}](./data/{parmlib}({member}).txt)'
        iplmap[parmlib][member]['from'] = f'{from_mem}'
        iplmap[parmlib][member]['content'] = dataset_read(parmlib, member)
    return iplmap

# Create local jcl
def create_jcl(jcl):
    my_rexx = '''/* rexx */
IsfRC = isfcalls( 'ON' )
address SDSF 'ISFEXEC SYSP'
do i = 1 to isfrows
    say PARM.i VALUE.i MEMBER.i REFNAME.i
end'''

    my_jcl=f'''//PROD001X JOB (124400000), 'REXX', CLASS=A,
//      MSGCLASS=X, NOTIFY=&SYSUID
//STEP1   EXEC PGM=IEBGENER
//SYSPRINT DD  SYSOUT=*
//SYSIN    DD  DUMMY
//SYSUT1   DD  DATA, DLM='@@'
{my_rexx}
@@'''

```

```

//SYSUT2 DD DSN=&&TEMPREXX(MYREXX),
//          DISP=(,PASS),UNIT=SYSDA,
//          SPACE=(TRK,(1,1,1)),
//          DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
/*
//STEP2 EXEC PGM=IKJEFT01,PARM='%MYREXX'
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSTSIN DD DUMMY
//SYSEXEC DD DSN=&&TEMPREXX,DISP=(OLD,DELETE)
/*
'''

    with open(f'{jcl}','wt') as o:
        o.write(my_jcl)

# Submit local jcl end return stepname - dataset output
def submit_local_jcl(jcl, stepname, dataset):
    job = jobs.submit(f'{jcl}', hfs=True)
    print('jobid: ',job.id)
    job.wait()
    print('rc: ',job.rc)
    if job.rc != '0000':
        output = jobs.list_dds(f'{job.id}')
        print(output)
        for item in output:
            stepname = item['stepname']
            dataset = item['dataset']
            print(jobs.read_output(job.id, stepname, dataset))
        exit(1)
    else:
        output = jobs.read_output(f'{job.id}', f'{stepname}', f'{dataset}')
    return output

# Functions End -----

# Main workflow

path = f'./data'
if os.path.exists(path) == False:
    os.mkdir(path)

# Get load_param_dsn, loadxx, ieasym_list, ieasys_list
out = oper_command('d','iplinfo')

```

```

# Retrieve values
data = {}
for line in out.split('\n'):
    # Get ieasym & ieasys lists
    if ' LIST = ' in line:
        key, value = line.split(' LIST = ', 1)
        data[key.strip()] = value.strip().strip('"')
    # Get loadxx & iplpar dsn from 'USED LOADPR IN SYS1.IPLPARM ON 04006'
    if 'USED LOAD' in line:
        words = line.split()
        loadxx = words[1]
        load_param_dsn = words[3]

# Extract, clean, and process the 'IEASYM' list
ieasym_list = data.get('IEASYM', '').replace('(', ' ').replace(',L)', ' ').split(',')
ieasys_list = data.get('IEASYS', '').replace(' (OP)', '').replace('(', ' ').replace(')', ' ').split(',')
if '00' not in ieasys_list:
    ieasys_list.append('00') # 'IEASYS00' always exists

# Downlad IPLPARM(loadxx) and create the iplmap dictionary
iplmap={}
iplmap['ds']=load_param_dsn
iplmap['loadxx']=loadxx
iplmap['path']=f'./data/{load_param_dsn}({loadxx}).txt'
iplmap['md']=f'[{load_param_dsn}({loadxx})](./data/{load_param_dsn}({loadxx}).txt)'
iplmap['content']=dataset_read(load_param_dsn, loadxx)

# Get PARMLIBs
parmlibs = zsystem.list_parmlib()

# Create dict for each parmlib
for parmlib in parmlibs:
    iplmap[parmlib] = {}

# Retrieve info for IEASYS and IEASYM
iplmap = find_members(iplmap, 'IEASYS', ieasys_list)
iplmap = find_members(iplmap, 'IEASYM', ieasym_list)

```

```

# Create temporary local jcl
jcl = 'temp.jcl'
create_jcl(f'{jcl}')

# Submit local jcl and retrieve output
stepname = 'STEP2'
dataset = 'SYSTSPRT'
output = submit_local_jcl(jcl, stepname, dataset)

# Retrieve info for PARMLIB members
parmlib_members = ['LNK', 'CMD', 'CON', 'LPA', 'OMVS', 'PROD', 'PROG', 'SCH']

lines = output.split('\n')
for line in lines:
    line = line[1:]
    words = line.replace('(', '').replace(')', '').split()
    if len(words) > 3 and words[0] in parmlib_members:
        list = words[1].split(',')
        iplmap = find_members(iplmap, words[3], list, words[2])

# Write the ipl_map dictionary to a JSON file
with open('ipl_map.json', 'w') as json_file:
    json.dump(iplmap, json_file, indent=4)

# Create a markdown file
markdown_content = '''---
markmap:
  colorFreezeLevel: 4
  initialExpandLevel: 4
  spacingVertical: 10
  spacingHorizontal: 80
  duration: 1000
---
'''

markdown_content += "# PARMLIBs and Active Members\n\n"
markdown_content += f"# Load: {iplmap['md']}\n"
for parmlib, members in iplmap.items():
    if isinstance(members, dict):
        markdown_content += f"## {parmlib}\n"
        for member, details in members.items():
            markdown_content += f"### {details['md']}\n"
            if details['from'] != '':

```



```

        markdown_content += f" - from.. {details['from']}\n"
    markdown_content += "\n"

with open('ipl_map.md', 'wt') as o:
    o.write(markdown_content)

# Get system symbols
out = oper_command('d', 'symbols')

symbols = {}
lines = out.strip().split('\n')

# Skip the first line and process the remaining lines
for line in lines[1:]:
    if '=' in line: # Check if the line contains a key-value pair
        key, value = line.split('=', 1)
        key = key.strip().rstrip('.') # Remove trailing period and
whitespace
        value = value.strip().strip('"') # Remove surrounding quotes and
whitespace
        symbols[key] = value

# Write the symbols dictionary to a JSON file
with open('symbols.json', 'w') as json_file:
    json.dump(symbols, json_file, indent=4)

```

The script execution time will depend on the amount of parmlib parameters selected, but in testings, it took no more than 10 seconds.

After successful execution there will be 3 files:

- Symbols.json: Python dictionary dumped as json file. We are not using it in this sample, but it can be helpful in case we need to resolve variables with symbols in the parmlib for example if we follow the IPL chain by reading the PARMLIB members:

IEASYSxx:

LNK=(AA,&ZVR,BB,L),

LNKLSTxx

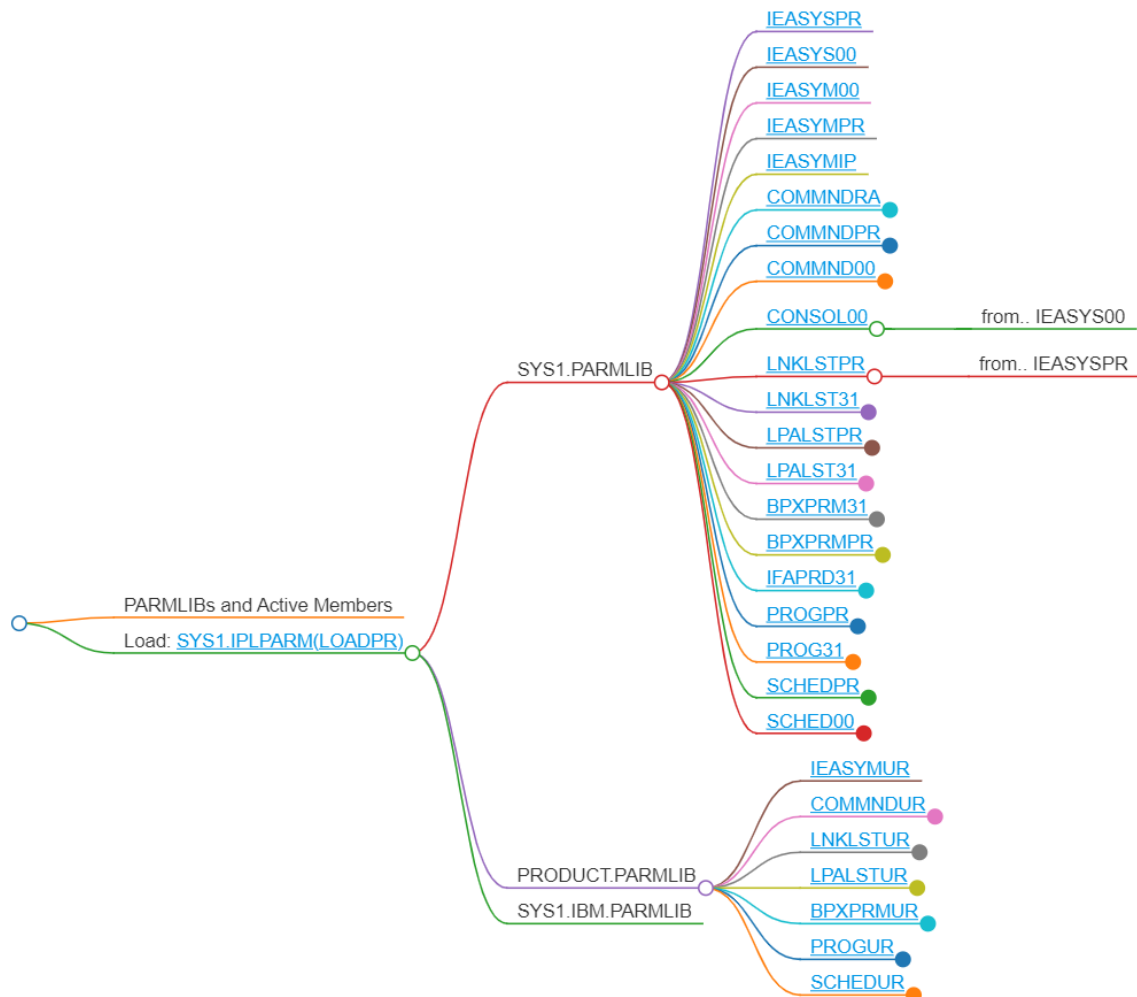
IEASYMxx:

SYMDEF(&ZVR='&SYSOSLVL(4:1).&SYSOSLVL(6:1)')

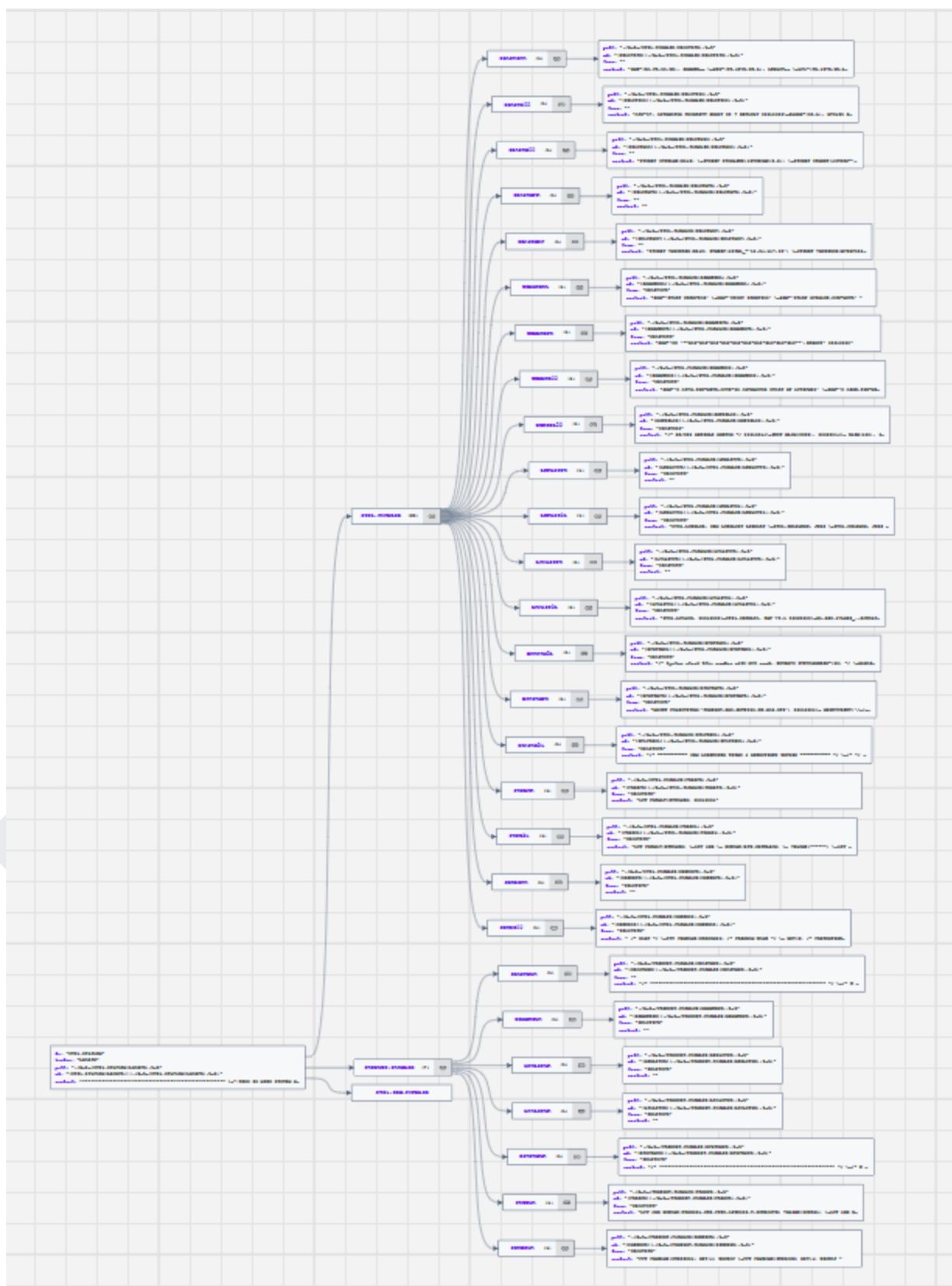
Symbol "&ZVR": "31" from the 'd symbols' command.

- ipl_map: Markdown Document. Use VSCode with Zowe Explorer and Markmap extensions to browse this file as a mindmap:

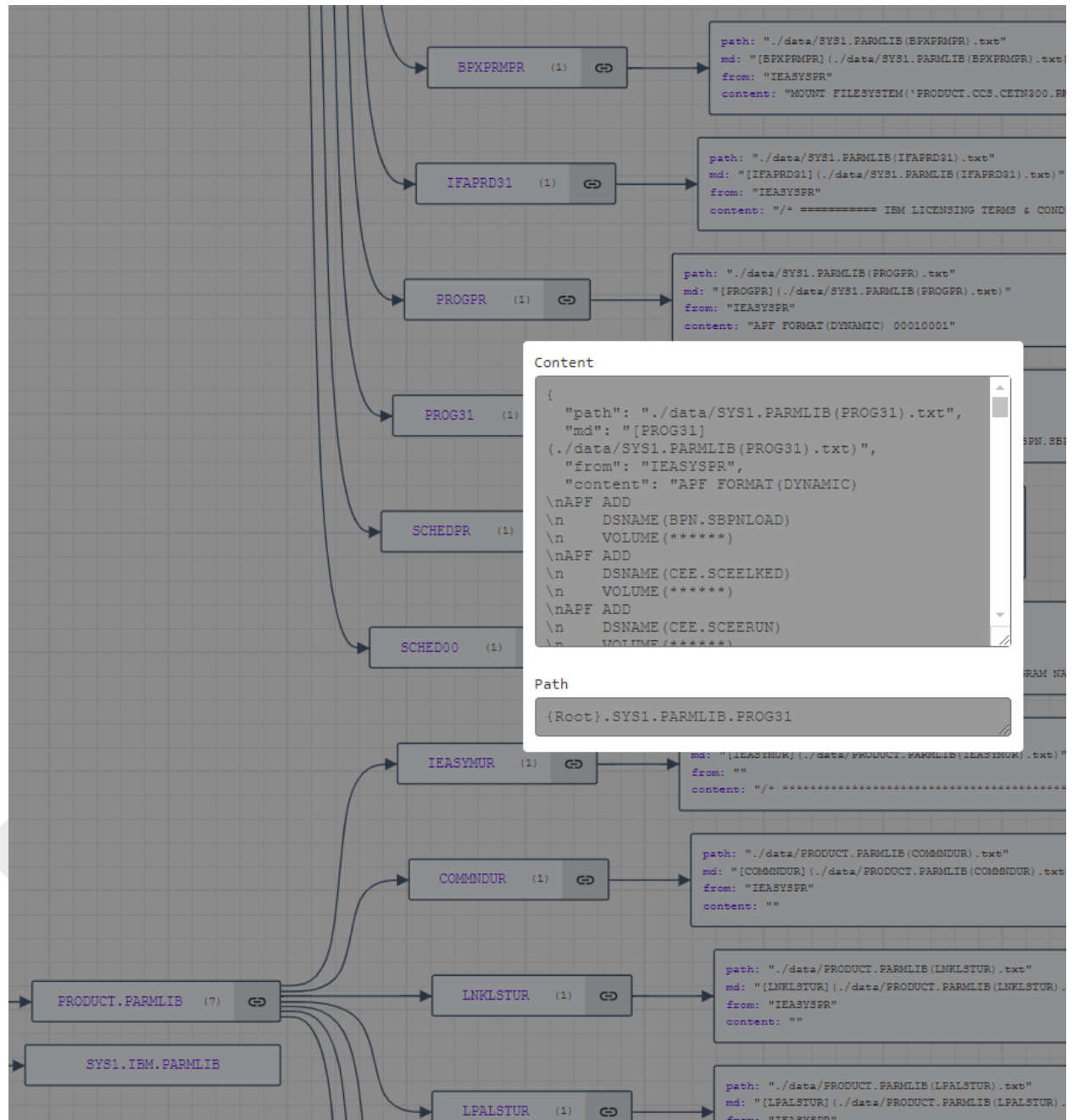
Clicking in the color dots will open the branch with the 'from IEASYSxx' information. Also the member hyperlinks will open in VSCode the members downloaded. To edit the real ones, use Zowe Explorer.



- Ipl_map.json: All information gathered from the execution is in a json file that can be used as input for further processing. Use VSCode and an extension called JSON Crack to see the IPL map in a graphical mode, and click on each member to see the content with more details than the md file:

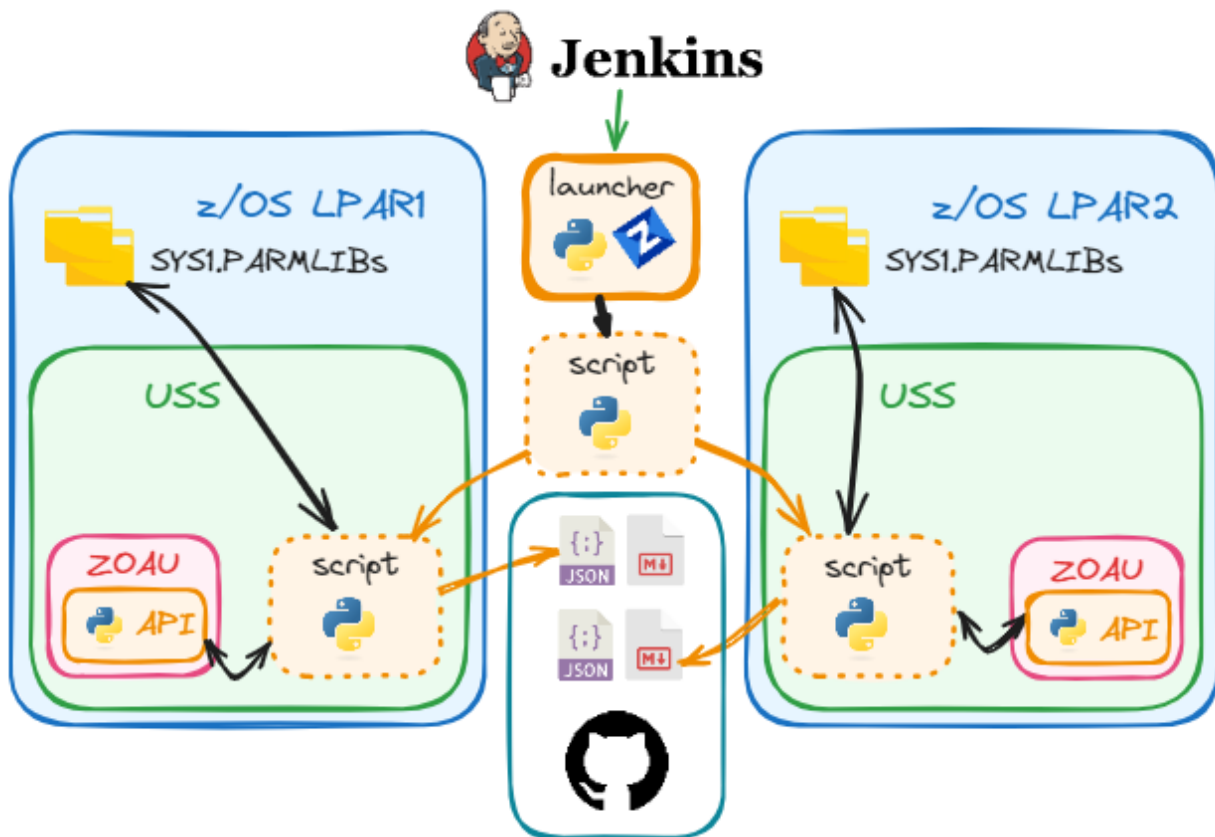


Detail for 'PROG31':



💡 Idea: A possible use case could be a script to check what is currently active in the LPAR (APF libraries, Mounted ZFSs, ...) and check if they are defined statically in the IPL concatenation PARMLIBs. Any dynamically added AFP libraries or Mounted filesystems are not added to the static configuration, it could cause problems for the next IPL.

IPL Map Launcher SSH & Zowe



Let's take one more step from the last case where we are generating a json file with the map of our IPL. If we want to generate independent maps, one for each LPAR, we need to run the `ipl_map.py` script individually in the target environments. If we make any modification in the script or we want to add more parms, it's ok while the script is somehow shared between the different USS. But what if we have isolated LPARs for different end customers or we don't share dasd for example?.

With an SCM tool we could deploy the script from a DEV environment to the rest of the LPARs by defining stages or PROD install LPARs and whatever method we have in place to send the file.

In this case, we are going to assume a scenario where a Jenkins Pipeline will loop through a list of LPARs and triggering a 'launcher' script that will execute the following procedure for each of them:

1. Connect to USS and get the 'home' directory
2. Delete a temporary directory 'temp_ipl' if exists
3. Create directory 'temp_ipl'
4. Upload our script 'map_ipl.py' to this directory
5. Execute the script on the target environment
6. Download all info generated (json, md files and PARMLIB members)
7. Delete the temporary directory

In this sample launcher script I use Zowe CLI for all steps except for step #5, where I run the script with SSH and explicit environment variables.

I use a different format of config file, in this case is a 2 level: one for lpar name and the second for the fields I need to ssh if I don't have Public/Private Key Authentication defined.

The config_multi.yaml file looks like this:

```
abcd:
  host: abcd.my_company.net
  Username: user
  password: pwd
dev:
  host: dev.my_company.net
  Username: user
  password: pwd
```

This script has to be executed passing the LPAR name argument in order to run (argparse Python library). This name has to match our Zowe Team configuration file for each LPAR (abcd.zosmf, dev.zosmf,...). At the end of every Zowe command I add `--zosmf-p {profile}.zosmf` or `--ssh-p {profile}.ssh` to target the desired LPAR:

```
Python launch.py -p abcd
```

```
# Script to launch the IPL map process in USS

import subprocess
import paramiko
import yaml
import argparse
import sys

try:
    parser = argparse.ArgumentParser(description='IPL Map ./data folder')
    parser.add_argument('-p', '--profile', help='Provide zosmf profile',
                        required=True)
    args = parser.parse_args()
except Exception as exc:
    print(sys.exc_info())
    print(exc)

profile = args.profile.lower()

zosmf_profile = f'--zosmf-p {profile}.zosmf'
ssh_profile = f'--ssh-p {profile}.ssh'

with open('config_multi.yaml', 'r') as f:
    confile = yaml.safe_load(f)

host = confile[f'{profile}']['host']
```

```

username = confile[f'{profile}']['username']
password = confile[f'{profile}']['password']

# Initialize the SSH client
client = paramiko.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

# Functions -----

def execute_command(command):
    try:
        result = subprocess.run(command,
                                shell=True, text=True, capture_output=True)
    except subprocess.CalledProcessError as e:
        print("Command execution failed.")
        print("Return Code:", e.returncode)
        print("Error:", e.stderr)
    return result.stdout, result.stderr, result.returncode

def execute_ssh_command(command):
    client.connect(host, port=22, username=username, password=password)
    exports = (
        "export
PATH=/usr/lpp/IBM/zooutil/env/bin:/usr/lpp/IBM/zooutil/bin:/usr/lpp/IBM/cyp/v
3r9/pyz/bin:/bin:/usr/lpp/java/current/bin:/usr/lpp/java/current"
        "export LIBPATH=/usr/lpp/IBM/zooutil/lib;"
    )
    remote_command = (exports + command)
    # Execute the remote command
    stdin, stdout, stderr = client.exec_command(remote_command)
    exit_status = stdout.channel.recv_exit_status() # Blocks until the
command finishes
    out = stdout.read().decode()
    err = stderr.read().decode()
    client.close()
    return out, err, exit_status

def disp_msgs(sto, ste, rc):
    print('sto: ', sto)
    print('ste: ', ste)
    print('rc : ', rc)
    exit(1)

def execute_zowe_command(zowe_command):

```

```

print(f'{zowe_command}')
sto, ste, rc = execute_command(zowe_command)
if rc != 0: disp_msgs(sto, ste, rc)
return sto

# Functions End -----

# Main workflow
if __name__ == "__main__":
    # Retrieve home directory

    # Get 'home' directory
    ssh_command = 'pwd'
    zowe_command = f'zowe uss issue cmd "{ssh_command}" {ssh_profile}'
    sto = execute_zowe_command(zowe_command)
    home = sto.split('$')[1].replace(' ', '').strip()
    print('home: ', home)
    if home=='/': home = '/tmp'
    # home = "/u/users/group/product/prod001/temp_ip1"
    temp_dir = f'{home}/temp_ip1'

    # Delete temp directory
    zowe_command = f'zowe files del uss "{temp_dir}" -rf {zosmf_profile}'
    # This script doesn't call the execute_zowe_command
    # because it fails if the directory doesn't exist
    print(f'{zowe_command}')
    sto, ste, rc = execute_command(zowe_command)

    # Create temp directory
    zowe_command = f'zowe files create dir "{temp_dir}" -m rwxrwxrwx
{zosmf_profile}'
    sto = execute_zowe_command(zowe_command)

    # Upload the ipl_map.py script to temp_ip1
    file = 'ipl_map.py'
    path = f'{temp_dir}/{file}'
    zowe_command = f'zowe files upload ftu "{file}" "{path}" {zosmf_profile}'
    sto = execute_zowe_command(zowe_command)

    # Execute script
    ssh_command = f'cd {temp_dir}; python {path}'
    print(f'{ssh_command}')
    sto, ste, rc = execute_ssh_command(ssh_command)

    # Cleanup temp artifacts

```



```

files = ['temp.jcl', 'ipl_map.py']
for file in files:
    path = f'{temp_dir}/{file}'
    zowe_command = f'zowe files delete uss "{path}" -fi {zosmf_profile}'
    sto = execute_zowe_command(zowe_command)

# Download Data,ipl_map.json, ipl_map.md and sysmols.json in zosmf_prof
folder
    folder = f'{profile}'
    zowe_command = f'zowe files download uss-dir "{temp_dir}" -d {folder}
{zosmf_profile}'
    sto = execute_zowe_command(zowe_command)

# Delete temp directory
zowe_command = f'zowe files del uss "{temp_dir}" -rf {zosmf_profile}'
sto = execute_zowe_command(zowe_command)

```

💡 Idea: I guess you have noticed a Github logo in the picture of this topic. We can commit and push the json files to a repo (on for each LPAR) whenever it detects any difference, so we could easily track changes and compare settings among different commit dates.

Again, this is the method I used for this use case, but I could have used others like Zowe SDK or z/OSMF REST API requests. Everything will depend on the goal and the resources available.

Regarding z/OSMF, my previous sample in the 'Python & z/OSMF' section showed a sample with the 'GET' method to retrieve a list of members and the content of one of them. To show other methods, here is a sample snippet on how to use z/OSMF with methods 'DELETE' and 'POST' and 'PUT' instead Zowe for the steps 2, 3 & 4:

```

import requests

# z/OSMF Configuration
base_url = "https://abcd.my_company.net:1443/zosmf/restfiles/fs"
headers = {
    'Authorization': f'Basic <your_credentials>',
    'Content-Type': 'application/json',
    'X-CSRF-ZOSMF-HEADER': ''
}

# File paths
remote_directory = f"/u/users/my_home/ipl_map"
remote_file = f"{remote_directory}/ipl_map.py"
local_file = "ipl_map.py"

# Step 2: Delete directory if it exists
def delete_directory(directory):

```

```

delete_url = f"{base_url}{directory}"
response = requests.delete(delete_url, headers=headers, verify=False)
if response.status_code == 204:
    print(f"Directory {directory} deleted successfully.")
elif response.status_code == 404:
    print(f"Directory {directory} does not exist.")
else:
    print(f"Error deleting directory: {response.status_code},
{response.text}")

# Step 3: Create the directory
def create_directory(directory):
    create_url = f"{base_url}{directory}"
    data = {
        "type": "directory",
        "mode": "RWXRW-RW-"
    }
    response = requests.post(create_url, headers=headers, json=data,
verify=False)
    if response.status_code == 201:
        print(f"Directory {directory} created successfully.")
    else:
        print(f"Error creating directory: {response.status_code},
{response.text}")

# Step 4: Upload the file
def upload_file(local_file_path, remote_file_path):
    upload_url = f"{base_url}{remote_file_path}"
    with open(local_file_path, "rb") as file_data:
        response = requests.put(upload_url, headers=headers, data=file_data,
verify=False)
        if response.status_code == 201:
            print(f"File {local_file_path} uploaded successfully.")
        else:
            print(f"Error uploading file: {response.status_code},
{response.text}")

```

GitHub repository

You can find this document and all sample code and data at:
https://github.com/drb1972/zos_python_integration.git

Draft