DWH Server Operation Manual

This manual is a resources aimed towards the project team, as well as future developer continuing the work of the DWH project.

In this documentation, the following operations will be addressed:

- SSH-Key
- SSH-Tunneling
- Portainer
- Cockpit
- Airflow
  - Accessing Airflow
  - Managing DAG with VSCode

SSH-Key

SSH-Keys are used to authenticate users on a server without using a password. Those keys come in pairs: a private key and a public key. The public key can be seen as the keyhole that's beeing installed on the server. The private key on the otherside is used by the users client to open the key hole. It is important to note, that the private key should be kept secret and secured by a strong passphrase. A passwordmanager like 1password, bitwarden or keypass is advised.

Creating SSH-Keys

The easiest way to create SSH-Keys is with UNIX commands (supported on MacOS and Linux) as well as GitBash Tool for Windows. We will now cover the generation of ED25519 keys:

```
$ ssh-keygen -t ed25519
```

Output:

```
Generating public/private ed25519 key pair.
Enter file in which to save the key ($HOME/.ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in $HOME/.ssh/id_ed25519.
Your public key has been saved in $HOME/.ssh/id_ed25519.pub."
```

Now that we have generated our keys, let's look at what's needed on the server. Every server has a directory for each user. Let's imagine, we have created a user named "Karl". The generated public key needs to be installed in to following location:

`/home/Karl/.ssh/authorized_keys`

The file `authorized_keys` is a textfile which has the following strucutre:

For example:

`ssh_rsa ASSDASDALASDFGDLSFGSDFGSDFGDFGSDFGSFGHJGHJç%345645645dfgvsdfGWç%TDSFGHASDFASDç*%*ç%DFS>YDFSASEFASDFASDFGHDSFGHDFGHSDAF"*ç"*ç%ç%&/ç%&ç%&*%&*ç% user@domain`

Remember, that this is your PUBLICKEY

Multiple SSH-Keys

It is possible to have multiple SSH-Keys installed. This section will briefly cover the usage of the users' personal SSH config file in order to maintain different SSH configuration for different hosts.

Linux / MacOS

On any Linux distribution, the ssh configuration file is stored under *.ssh/configuration*. To access the file via the terminal type `$ vim ~/.ssh/configuration` or open it with any text editor.

The SSH configuration file has many different attributes. We will cover the most important ones in this example.

```
# GITHUB
Host github.com
    user git
    HostName github.com
    PreferredAuthentications publickey
    IdentityFile ~/.ssh/id_rsa`

# GITLAB
Host git.example.ch
   User git
   HostName git.example.ch
   PreferredAuthentications publickey
   IdentityFile ~/.ssh/id_ed25519_lab
   Port 20730


# SOMEOTHER DOMAIN
Host <some_ip or hostname>
    User remo
    HostName test.aiforge.ch
    IdentityFile ~/.ssh/id_rsa_aiforge
```

The most important entries in this file are the domain and the identity file location. With these entries, SSH picks automaticly the correct key and passes it to the server for authentication. If there is no file specified, depending on the SSH daemon, SSH will try default key locations and default names (id_rsa for example).

Note: There are many different key encryption types available. Some provider have restrictions on the encryption type of the SSH-Keys.

GitLab reccomnends ED25519 encryption (Use SSH keys to communicate with GitLab | GitLab)

Test SSH Connection

After you setup your SSH configurations, it is adivsable to test your SSH connection. You can do this with the following command:

`ssh -T user@domain.ch`

This command will establish a connection and terminates it imidiately. If there is an error, try to use:

`ssh -vvv user@domain.ch` and copy the output. Your sys admin (me) will need it to find the problem (usualy file permissions).

SSH-Tunneling

In order to understand SSH-Tunneling, we need to know some things about firewalls. A firewall manages traffic going through the different ports on your system. In order to access a network (or a Linux instance) throgh SSH, port 22 must be open. There are some cases, where the SSH port is on a different port than on 22. This is a good idea security wise, since hackers are always aiming towards port 22. In some circumstances, we even honeypot port 22 (send gibberish as a response), just to mess with hackers. In general, SSH Ports are secured with special tech such as SELinux. Password login for SSH should **always** be turned off since this is a substancial security issue.

Imagine we have a system with only port 22 (SSH) open. All the other systems like Airflow, Portainer, Cockpit are on different port which we cannot access (since only port 22 is open). What we can do now is SSH into the system with port 22 and "bind" the localhost port to our own port. SSH does this by compressing the connection into the SSH port forwarding. This is a very secure way to access the ports. It's only downside is, that this method should only be used with stable network connections (which are always provided by switch).

In order to SSH-Tunneling, we do the following:

```
$ ssh user@ip-address -L [CLIENTPORT]:[localhost OR ip-address]:[target_port]
$ ssh centos@ip-address -L 9090:localhost:9090
```

The terminal will keep an open terminal with the active SSH connection. If you open your browser now and check for localhost:9090, you should now see "what the server sees on port 9090". In our case, we have the app "Portainer" running on port 9090, which is also accessible through the public web (SSL secured).

What programms are we accessing with SSH-Tunneling in our project?

Mainly, SSH-Tunneling is used for accessing Airflow, which runs on port 8080.

Congrats! You have now mastered SSH :-)

Portainer

Portainer is a graphical management application for docker containers. It is very helpful to monitor, deploy and edit container. Since we are runnign Airflow, scrapers and Flask in docker containers.

Generally, it is good practice to deploy containers with docker compose via the terminal and the corresponding docker-compose.yaml files.

Our Portainer is running on https://test.aiforge.ch:9443

Credentials are provided in a different channel.

Cockpit

Cockpit runs on https://test.aiforge.ch:8080 and is used as a monitoring platform for Linux. It's main purpose is to monitor access and ressources.

Airflow

Airflow is an ETL tool which is used to automate scrapers, cleanup and preprocessing.

Overview

Airflow is deployed in a docker container on the server. Usually, docker containers are closed systems and are non-persistent. In order to achieve persistency, docker volumes are created and "attached" to the container. Further, ports of the container must be "bound" to the ports on the system in order to be accesible for users. Generally, docker containers are very hand to work with since they have their own dependencies. The downside to these containers are, that theiy are difficult to troubleshoot if something should not work as intended.

Docker Setup

Airflow is running inside a Docker setup. This specific setup has several advatages, but also disadvantages. For one we can create a reproducable installation of airflow and have all dependancies contained within a container. This is important since we deploy more than just airflow on our server. The negative aspect of docker is, that we have to deal with an additional piece of technologie to manage. Since Docker is completely seperate from the host system, we need to create several volume bridges in order to enable airflow to communicate with our host server. Further, the packages in airflow are predefined. That means, that it is not an easy task to add packages on the fly.

In order to achieve persistency, we defined the following volumes to act as bridges between the hosts system and the airflow containers:

- /opt/airflow_container/dags
- /opt/airflow_container/logs
- /opt/airflow_container/scripts
- /opt/airflow_container/data
- /opt/airflow_container/plugins

In order to start airflow, the following command needs to be run in the shell with the working directory set to /opt/airflow_container:

```
docker-compose up
```

This will initiate the custome Dockerfile and the custom docker-compose.yaml file and start the airflow containers.

Python Packages

In order to add packages to the airflow container, the requirements.txt file needs to be adapted. Upon startup, docker-compose will read the requirements file (by building the image with the Dockerfile) and installs all the packages with the pip install command.

Writing DAGs

DAGs are one of the core elements of Airflow. They schedule jobs and can have their own logic elements built into them. DAGs are written in python and deployed on Airflow.

DAGs are located in to following location on the server:

`/opt/airflow_container/dags`

In this folder, we find our own dags which can later be deployed on the web application of ariflow.

Note: Airflow is installed in the /opt/ folder. This means, that sudo privileges are required to write in this folder. There are two ways we can do this:

- use `sudo vim somename.py` and edit the script directly in the terminal
- use VSCode extension to save as sudo

The later is advisable since VSCode has a language server for python installed and is generally more accessible to users than VIM.

In order to save as root in VSCode, we need two extensions:

1. Remote - SSH
2. Save as Root in Remote - SSH

With these tools, we are able to write code in VSCode and save them diretly to the DAG folder.

Another optional but reccomended extension is the SSH -FS. Once configured, it enables the user to open a file system like on the local machine.

The following must be specified in order to use SSH - FS:

- Host (ip-address)
- Port = 22
- Root = /opt/
- Username = Karl (or your username)
- Private Key = $HOME/.ssh/id_rsa
- Passphrase

Python Operators

The easiest language to write DAGs is tu use simple python syntax. Best practice is to write a callable function and then use this callable in the following syntax:

```python
from airflow.models import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime

## some logic function

def callable1():
    return "some logic"

def callable2():
    return "some other logic"

with DAG("flair_dag",
        start_date=datetime(2022, 12, 1),
        schedule_interval="@daily", catchup=False) as dag:

        task_2 = PythonOperator(
            task_id = 'task_1',
            python_callable = callable1
        )

        task_2 = PythonOperator(
            task_id = 'task_2',
            python_callable = callable2
        )

        task_1 >> task_2
        ## task_1 is run before task_2
```

Bash Operators

The second operator used in this project is the BashOperator. We use it to clean up files we no longer need or the push data into archive folders. We can run shell-scripts or write directly into the bash_command variable:

```python
from airflow.models import DAG
from airflow.operators.bash_command import BashOperator
from datetime import datetime

with DAG("flair_dag",
        start_date=datetime(2022, 12, 1),
        schedule_interval="@daily", catchup=False) as dag:

        task_1 = BashOperator(
            task_id = 'task_1',
            bash_command = "ls | grep *csv | mv archive/"
        )

        task_2 = BashOperator(
            task_id = 'task_2',
            bash_command = "rm *.txt"
        )

        task_1 >> task_2
        ## task_1 is run before task_2
```

Acessing Airflow Web

Airflow web can be accessed by SSH-Tunneling and mounting Port 8080. Since the application is secured by SSH already, username and password are not really needed.

username: airflow

password is provided on different channel.

Have Fun :)