Part I – Answers for Debug Systems Issues

Question 1

Apparently there is a difficulty connecting to the DB service/server. In order to resolve this, I would choose to follow the steps below:

- Backup the database;
- Check the availability of the DB service/server;
- make the DB service available (If the above is not complied with);

Question 2

Considering the infrastructure and the error presented:

1. Everything indicates that the tomcat service is down. My first action would be to backup the application's database and then `start` the tomcat service to view the logs, if it doesn't start.

2. The following could be the possible issues:
    - **Tomcat Stopped**: Solution would be to start the service.

- **Tomcat Server Overloaded:** restart the service and install a monitoring tool on the infrastructure (eg: *munin*) to identify the real cause in the next times.
- **Connection problems between Reverse Proxy and Tomcat:** check if `.conf` file has been configured correctly.
- **Blocking the listening port of the tomcat container/service:** enable the port on which the service runs.

Question 3

I believe it's necessary to add variable definition (as a function) to referenced file, then use in tags.tf appropriately.

Example could be:

```
 variable "ENV" {
    type = string
    default = "default-value"
}
```

Question 4

My first action would be to try to resetting the master user, in order to recover the writing privileges and assign them to the users according to the context.

More details on support would find here, at official support platform.

Part II – Linux Laboratory

> The IP Address my change, depending on network you are connected. Use `ip a` to check the IP assigned to your VM copy.

Virtual Machine and Operating System

I used virtualBox version 6.x, where I created a virtual machine and installed the Operating System as proposed. Also made sure that all packages are up to date, installed *nano* , *openssh*, *ufw* and set *Bridged Adapter* as the network type for the VM.

> **NOTE:**
>
> > *Security settings such as changing the default ssh port, creating users with limited privileges, login by RSA key pair, etc, were not considered, assuming that is not what is being evaluated (but recognizing the need).*

Using the following command:

```
scp wit-cicd-challenge.jar wit@192.168.31.12:/home/wit/
```

I ensured that the **.jar** file was loaded from my machine (windows 11) to the VM.

User and privileges

These commands were executed to create `wit` user and add him to sudo group:

```
sudo adduser wit
sudo passwd wit
```

```
sudo usermod -aG wheel wit
```

To test its operation, just execute `su wit` to login using wit user.

Now we have the wit user created and with the necessary privileges to move forward.

Docker installation and configuration

> *Instructions provided*, at the link:https://docs.docker.com/engine/install/centos/

```
sudo yum install docker
```

In order to run docker without sudo, it was necessary to create a group and associate the user in order to have the necessary privileges.

```
sudo groupadd docker
```

Then, adding the user by running:

```
 sudo gpasswd -a $USER docker
```

Testing, it was noted that the configuration was successful.

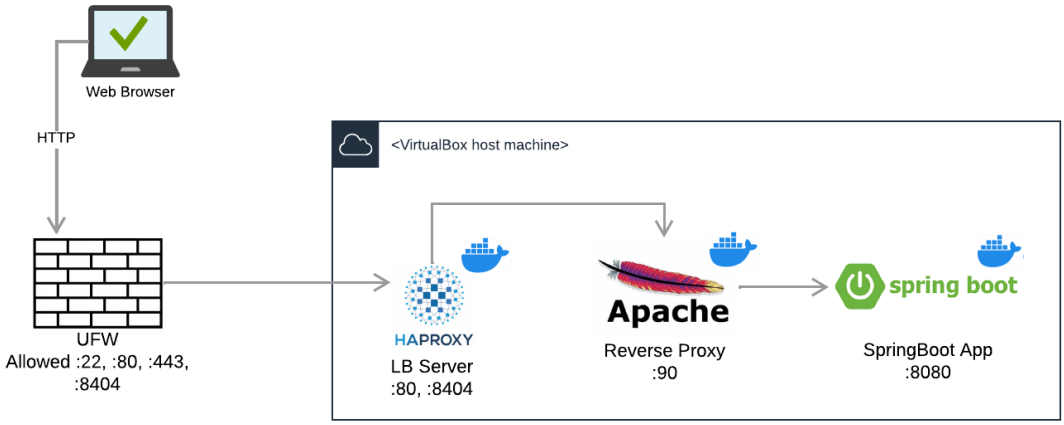
Architecture and description of the proposed scenario

The image below illustrates the scenario I configured, upon request.

Specifically, it is a network of containers connected to each other in order to send requests and responses between them.

There are three (3) containers:

- The first for the *Load Balancer*,
- The second for the *Reverse Proxy*, and
- The third, for the Spring Boot application.



The following tools were chosen/used:

	Tool	Comments
Containers	**Docker**	-

	Tool	Comments
Load Balancer	**HAproxy**	First layer, in contact with the outside, serving on port **:80**
Reverse Proxy	**Apache**	-
Firewall	**UFW**	To ensure that only the LB is accessed from the outside, the others will be accessed from the other containers or *host*

Network Configuration

Before starting with the creation of containers, I created a bridge network to later connect all the containers that are created. The following command was used to create the network named **redewit**:

```
docker network create --driver=bridge redewit
```

Executing `docker network ls`, it was possible to confirm the existence of the previously created network.

Spring Boot Container: Creation and Configuration

For organizational reasons, I created folders to organize the files related to each container. The container associated with the Spring Boot application will be called *wit-test*, so the folder created also has the same name.

```
cd ~
```

```
mkdir wit-test
```

```
cp wit-cicd-challenge.jar wit-test/
```

```
nano wit-test/Dockerfile
```

I also created it in the folder or file named **Dockerfile** and copied the following content:

```
FROM openjdk:11
COPY wit-cicd-challenge.jar wit-cicd-challenge.jar
ENTRYPOINT ["java", "-jar", "/wit-cicd-challenge.jar"]
```

- Where:
  - **Openjdk:11** is the official image created by docker
  - On the second line the **COPY** statement specifies that the .jar file should be copied

- Finally, **ENTRYPOINT** specifies the command to be executed to host the application when the conainer is created.

Then I ran the following command to create a Docker image for the current Spring Boot project:

```
docker build -t wit-test wit-test/
```

Note that the first parameter refers to the image name and the second to the folder where you should find the files to be used for the build.

After executing the last command, it is possible to view the images in question using the `docker images` command.

Now there is only the image that is ready to be used in the creation of the container. The following command will create the container, allow it to be visible/accessible from the outside on port **:8080** and also ensure that it is the service that will start with the operating system:

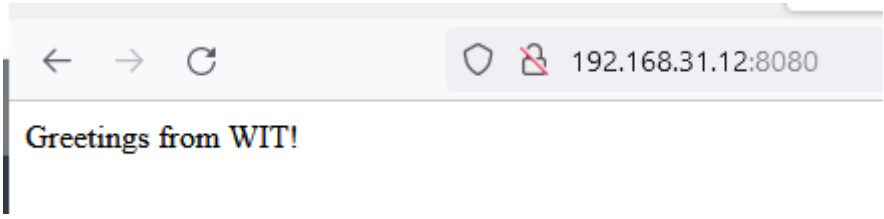
```
docker run -d --restart unless-stopped -p 8080:8080 --net redewit --name wit-
test wit-test
```

We can confirm by running `docker ps` the existence of the container and its details.

At this point, we can already visualize the result from the outside (browser of our host machine that is running the VM):

> `<ip-do-seu-servidor>:8080`

The result should be as shown bellow:



Reverse Proxy: Creation and Configuration

Now that I've configured the application's container, I'm going to configure the reverse proxy that will forward traffic to the application.

First of all, I used the `docker pull httpd:latest` command to pull the image from the latest version of httpd. In the proxy folder, I created the **proxy** directory to contain the **Dockerfile** and related files.

```
mkdir proxy
```

```
nano proxy/Dockerfile
```

The file Content:

```
# The Base Image used to create this Image
FROM httpd:latest

# to Copy a file named httpd.conf from present working directory to the
/usr/local/apache2/conf inside the container
COPY httpd.conf /usr/local/apache2/conf/httpd.conf

# This is the Additional Directory where we are going to keep our Virtualhost
configuraiton files
RUN mkdir -p /usr/local/apache2/conf/sites/

# To tell docker to expose this port
EXPOSE 90

CMD ["httpd", "-D", "FOREGROUND"]
```

Still in the created **proxy** folder, I created the configuration file `nano proxy/httpd.conf` with the content from that link (Apache2 Conf File).

Build and creation of the image from the **proxy/Dockerfile** file will be performed after executing the command:

```
docker build -t proxy proxy/
```

and the same is named *proxy* and can be confirmed by running `docker images`.

Here's the creation of the workspace that will be used to *mount* in the container and will contain some configuration files. There are 2 directories, where the first one stores the `.conf` files and the second the `.html` files.

```
mkdir -p /home/wit/apps/docker/apacheconf/sites
```

```
mkdir -p /home/wit/apps/docker/apacheconf/htmlfiles
```

Now the creation of the *.conf* file named *demowit.conf*:

```
nano /home/wit/apps/docker/apacheconf/sites/demowit.conf
```

to contain the following content:

```
 <VirtualHost *:80>

    ServerName demowit.local
    ServerAlias www.demowit.local

    ServerAdmin exemplo@demowit.local
    DocumentRoot /usr/local/apache2/demowit

    <Directory "/usr/local/apache2/demowit">
        Order allow,deny
        AllowOverride All
        Allow from all
```

```
        Require all granted
    </Directory>

    ErrorLog logs/demowit-error.log
    CustomLog logs/demowit-access.log combined

    ProxyPass / http://wit-test:8080/
    ProxyPassReverse / http://wit-test:8080/

</VirtualHost>
```

Now the creation of the `.html` file that will serve as the landing page:

```
nano /home/wit/apps/docker/apacheconf/htmlfiles/index.html
```

The content:

```
<html>
    <head>
        <title>demowit</title>
    </head>
    <body>
        <h2> Funcionando Perfeitamente... </h2>
    </body>
</html>
```

The last configuration for this step is the creation of the container, associated with the publication of the port and the *mount* of the directories/files to be used in the container, by using the command:

```
docker container run --publish 90:80 -d --restart unless-stopped --name proxy --
net redewit -v
/home/wit/apps/docker/apacheconf/sites:/usr/local/apache2/conf/sites -v
/home/wit/apps/docker/apacheconf/htmlfiles:/usr/local/apache2/demowit proxy
```

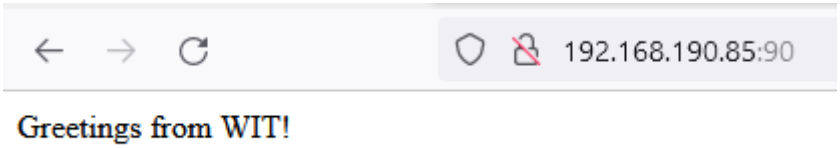
Remembering that I configured port **:90** for the reverse proxy.

In the */etc/hosts* file of the host machine, the localhost IP must be associated with the local DNS that is being used in the configuration files:

```
127.0.0.1           demowit.local
```

> **To test this configuration:**
>
> > `<ip-do-seu-servidor>:90` in the browser, outside the server and the same network, and `curl demowit.local:90/` inside the server (command line).

On the **/** route, the application that we configured earlier is running, from the proxy container:



> We have configured the proxy, the intermediary between the LB Server and the Spring Boot Application.

Load Balancer: Creation and Configuration

I used **HAproxy** as the Load Balancer. First step was to create the configuration file, to configure the operation of the container, for that I used the command:

```
nano haproxy.cfg
```

The following code was included in the file:

```
global
  stats socket /var/run/api.sock user haproxy group haproxy mode 660 level admin
expose-fd listeners
  log stdout format raw local0 info

defaults
  mode http
  timeout client 10s
  timeout connect 5s
  timeout server 10s
  timeout http-request 10s
  log global

frontend stats
  bind *:8404
  stats enable
  stats uri /
  stats refresh 10s

frontend myfrontend
  bind :90
  default_backend webservers

use_backend app-b if { path /wit-test } || { path_beg /wit-test/ }

backend webservers
  server s1 proxy:80 check

backend app-b
  http-request replace-path /wit-test(/)?(.*) /\2
  server s2 proxy:80 check maxconn 30
```

the LB container will serve on port **:90** and traffic from port **:80** of the host machine will be redirected to port **:90** of the LB container.

The HAproxy dashboard will also be available on port **:8404**, for management.

The `app-b` is also pointing to the `proxy:80`, it's a way to include /wit-test route. Then, **/** and **/wit-test** are pointing to same app.

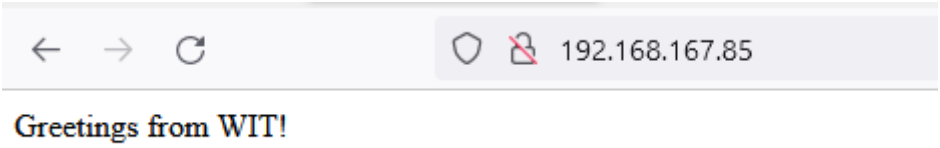
Being in the directory where we created the configuration file, I executed the following command, to create the container, configure the port rule and the volume mount of the file used.

```
sudo docker run -d \
   --name haproxy \
   --net redewit \
   -v $(pwd):/usr/local/etc/haproxy:z \
   -p 80:90 \
   -p 8404:8404 \
   --restart unless-stopped \
   haproxytech/haproxy-alpine:2.4
```

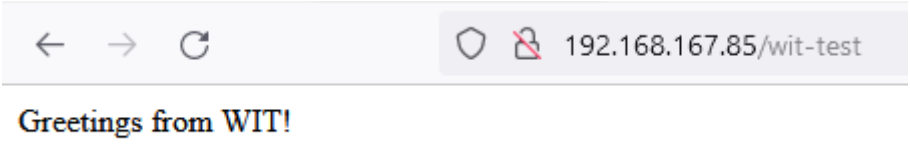
Once this is done, the configuration has been successfully completed, so it can be tested..

> **To test this configuration:**
>
> > `<ip-do-seu-servidor>` in the browser, outside the server and the same network, and `curl demowit.local` inside the server.

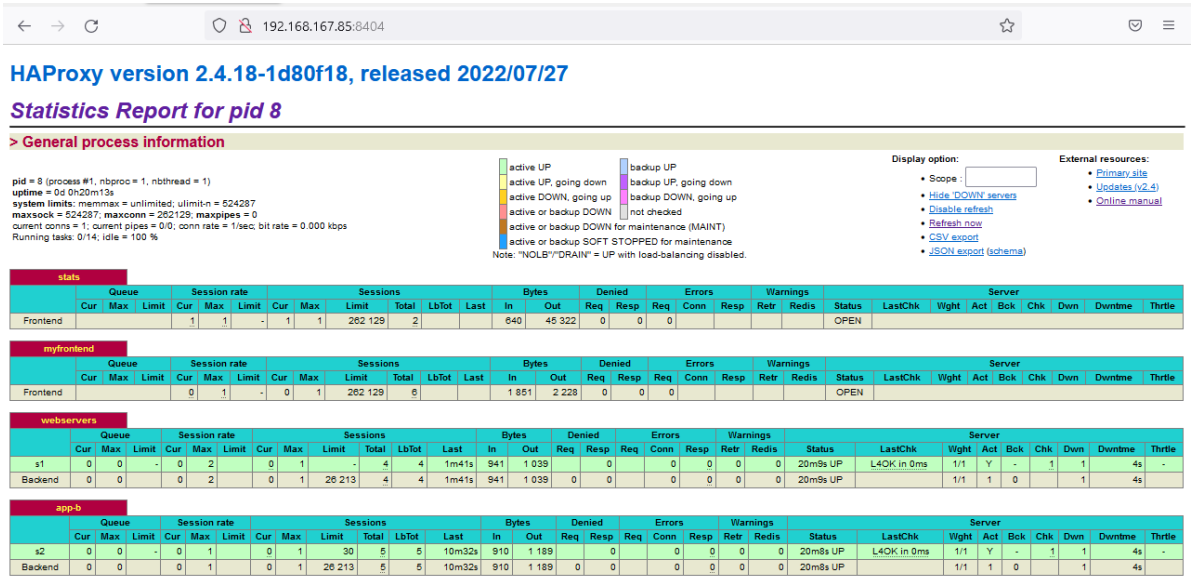
On the **/** route, from the **:80** port, the application that we previously configured is running, from the proxy:



On the **/wit-test** route, from the **:80** port,



In the **/** route of the **:8404** port, returns the haproxy dashboard:



General configurations

Having the whole flow working, I activated the firewall to ensure that access will only be from port **:80**, **:443**, **:8404** for http and port *:*22 * for SSH.

```
sudo ufw limit 22/tcp
```

```
sudo ufw allow http
```

```
sudo ufw allow https
```

```
sudo ufw limit 8404/tcp
```

```
sudo ufw enable
```

Conclusion

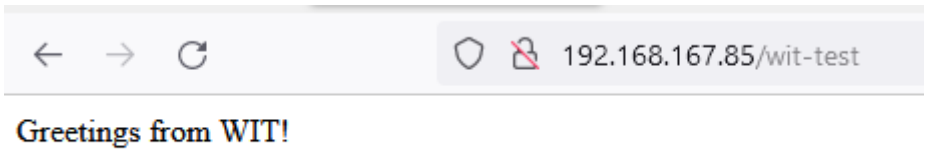
The containers created, as proposed:

```
[wit@localhost ~]$ docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED          STATUS          PORTS                                                NAMES
a850a0ca1a60   haproxytech/haproxy-alpine:2.4  "/docker-entrypoin..."   4 minutes ago    Up About a minute   0.0.0.0:8404->8404/tcp, 0.0.0.0:80->90/tcp   haproxy
16f3d8daa50e   proxy                          "httpd -D FOREGROUND"    40 minutes ago   Up About a minute   90/tcp, 0.0.0.0:90->80/tcp                   proxy
418f68e2bf28   wit-test                       "java -jar /wit-ci..."   About an hour ago   Up About a minute   0.0.0.0:8080->8080/tcp                   wit-test
[wit@localhost ~]$
```

Executing `curl http://demowit.local/wit-test/` the result is shown bellow:

```
[wit@localhost ~]$ curl http://demowit.local/wit-test/
Greetings from WIT![wit@localhost ~]$
[wit@localhost ~]$
```

Also from outside, using /wit-test route:



Greetings from WIT!