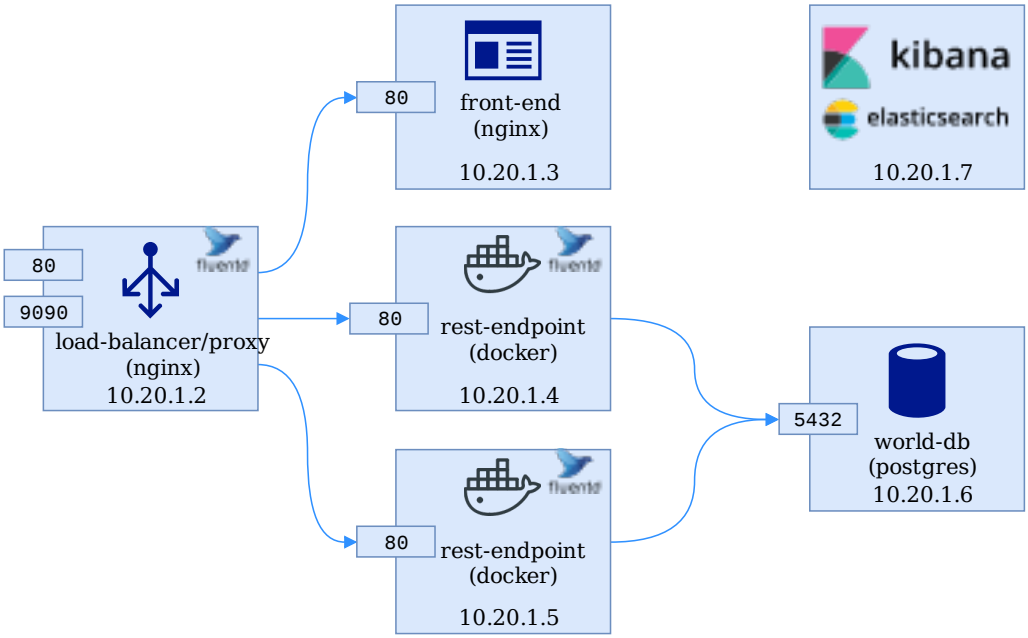
Lab 7

More complex playbooks

During this lab you will create an ansible orchestration to roll-out the full stack of the "World App". You have already worked on some pieces but during this lab you will improve those plays with newly learned concepts and add new plays. You've seed the architecture diagram below during an earlier lab:



Your working directory for this lab is `~/course/ansible/worldstack`

Start by copying your `worlddb.yaml` from `~/course/ansible/worlddb` to `~/course/ansible/worldstack` (you don't have to copy the `structure.sql` as we have already placed that inside the `files` directory for you.

You will improve this playbook a little by introducing a condition and you will get rid of the clear text password

Only reload when needed

At this moment, every time you run your playbook, the postgres daemon gets reloaded. But in fact this is only needed when the when you changed the service's `pgdata.conf` . This is the code you have:

```
- name: Ensure PGDATA is set for the postgres service
  ini_file:
    path: /etc/systemd/system/postgresql.service.d/pgdata.conf
    section: Service
    option: Environment
    value: PGDATA={{pgdata_dir}}
    create: yes
- name: reload the daemon
  systemd:
    name: postgresql
    daemon_reload: true
```

As you know modules are therefore tasks are idempotent. So perhaps the `pgdata.conf` does not need updating, so a reload is not required. Let's improve this design

Make sure the reload only occurs if the postgres service has been updated use `register` / `when`

Using a Vault

In a previous lab you entered the password clear text for our database. Let's encrypt this using a vault:

- Create a secrets file named `vars/secrets.yaml` `ansible-vault create vars/secrets.yaml`
- Set the password to something you will remember (e.g, *tiger*)
- add a variable named `worlddb_password` with `masterkey` as its value this is a yaml file

Check the file contents and notice it is encrypted:

```
$ cat secrets.yaml
$ANSIBLE_VAULT;1.1;AES256
653331616665386266646231383363386236646561306630663530623237636664 3
…
376335363365653461353333636462353635633337616264663265396635663131 62
```

View the unencrypted contents of the file `ansible-vault view …` to double check your password and the value of your variable

Then change your playbook so that it:

- includes the vars into your play `var_files`
- no longer uses the clear-text password but uses the value from the vault variable `{{worlddb_password}}`

To test you playbook make sure you make ansible ask for your password `--vault-id @prompt`

Open the empty `rest-server.yaml` . You will use this file to provision and deploy the country rest service endpoint on two nodes. This service is build using Spring Boot (https://spring.io/projects/spring-boot). The service will use our database. The application has been containerised, so you will need to run a docker container. The image is available on docker hub edc4it/rest-countries (https://hub.docker.com/r/edc4it/rest-countries/).

Start with the basics:

- This play should be ran against `rest-servers`
- Elevate the privileges to root

Install Docker

Then we are going to install Docker CE. Some of the installation information is available on docs.docker.com (https://docs.docker.com/install/linux/docker-ce/centos/#set-up-the-repository).

- First we need to register the official docker CE yum repository. If you have a look at https://download.docker.com/linux/centos/docker-ce.repo (https://download.docker.com/linux/centos/docker-ce.repo), you'll notice the `docker-ce-stable` repository. Use ansible to register a repository with

these values. You might want to consult the yum_repository (https://docs.ansible.com/ansible/latest/modules/yum_repository_module.html#yum-repository-module) documentation.

- After that install the `docker-ce` package
- Then enable and start the docker service

Install python dependencies

Later below you will use ansible docker modules. These modules require the *python* package `docker-py` . Use the correct module to install this library pip

Start a docker container

You will be using the docker-container (https://docs.ansible.com/ansible/latest/modules/docker_container_module.html#docker-container-module) module. Have a look at the documentation for this step.

Here is some useful information:

- Name the container anything you want, for example `rest-countries`
- The name of the image is `edc4it/rest-countries:1.4`
- Make sure it is started
- Publish the port 80 `published_ports:` with `- "80:80"` (don't forget the quotes)
- Make sure the container is always restarted in case in fails `restart_policy: always` (this also makes sure a container starts when the machine starts)

You will also need to configure the container to use our database: The Spring boot application inside the container has a configuration file, which you can find on gitlab application.yaml (https://gitlab.com/edc4it/docker/rest-countries/blob/master/src/main/resources/application.yaml).

Notice the following three configuration parameters:

- `spring.datasource.url` which is a jdbc *connection* URL to the database
- `spring.datasource.username` / `spring.datasource.pasword` the credentials used to connect to the database.

These values can be overridden in various ways, one of which is through environment variables. This is explained below, first you'll set the database connection url and then the credentials.

Database Connection URL

- You will need to obtain the host variables of the machine running our database so that you can get the ip address (the path to this variable is `['ansible_eth1']['ipv4']['address']`). Do you recall can you reference the host variables of another host? `hostvars[…]`
- What is the name of the host (hint have a look at `~/course/ansible/setup/hosts`) `master-db-server`
- Use this to create the following connection url for `spring.datasource.url` : `jdbc:postgresql://…/worlddb` (replace the three dots with your jinja2 expression)
- Now important: because you are referencing a host inside the inventory which is not included in this play, you will need to add another play to this playbook with just a reference to the hosts:

```
- hosts: db-servers # add this line
- hosts: rest-servers
  become: true
```

Database Credentials

Now for the database username and password. The password is available as a variable inside the vault, we could have also setup a variable for the username, but we haven't. For the username (`spring.datasource.username`) just use the literal value `student` .

Where do you have the password available, again? Yes inside the `vars/secrets.yaml` vault file

- Include the variables inside this file to your play `vars_files`
- Set `spring.datasource.password` using a variable expression to your password variable `{{worlddb_password}}`

Test

Run your playbook (don't forget to make it ask for the vault's password). Please recall you have two machines inside the rest-servers inventory group: `10.20.1.4` and `10.20.1.5`

You can use the one of the swagger UIs (http://10.20.1.4/swagger-ui.html (http://10.20.1.4/swagger-ui.html) and http://10.20.1.5/swagger-ui.html (http://10.20.1.5/swagger-ui.html)) to test the service or use cURL (and optionally pipe it through jq (https://github.com/stedolan/jq)). To get for example all countries in Africa (the `continent` parameter is required):

```
$ curl -s http://10.20.1.4/countries?continent=africa | jq .
```

When you see a list of countries, then you know the application was able to connect to the database (it might take some time as the Spring Boot application needs to start, you can check the logs if you are in doubt or if there is an error)

Open `~/course/ansible/worldstack/load-balancer.yaml` . Notice we have already added three plays. The first two are there so that you can access the host variables of machines in those groups (as you have done yourself in the previous step)

First thing is to install nginx as you've done before for the front-end:

- For the name choose `nginx-repo`
- a description "Official nginx repository"
- the url is `http://nginx.org/packages/centos/$releasever/$basearch/`
- and make sure to disable the `gpgcheck`

Then Install the following packages:

- `nginx-1.14.0`
- `ca-certificates`
- Don't start the `nginx` service yet!

SELinux

The rest endpoint will be on port 9090. By default selinux does not allow this. There is a ansible module named seport (https://docs.ansible.com/ansible/latest/modules/seport_module.html) with which we can add a SELinux port type definition. This module does require the `policycoreutils-python` package to be installed:

- Use a *separate* task to install this `policycoreutils-python` package (*don't* add it to the list of packages you already have in place for `nginx-1.14.0` and `ca-certificates` . You'll understand later why we ask you to use a separate task.)
- Then use seport (https://docs.ansible.com/ansible/latest/modules/seport_module.html):

parameter	
ports	9090
proto	tcp
setype	http_port_t

- After this make sure nginx is started and enabled

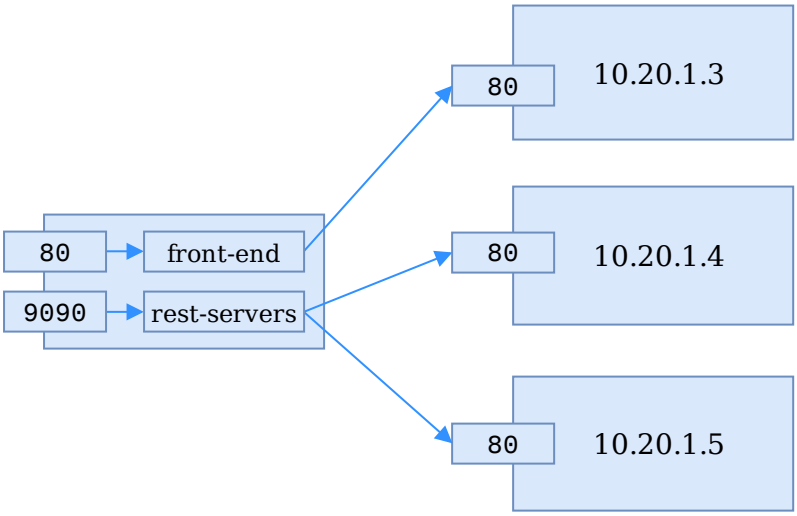
nginx configuration overview

You will need to add configuration for nginx and set it up as a reversed proxy/load-balancer.

First you want to make sure the default server is unavailable

- Use the correct module to delete the file `/etc/nginx/conf.d/default.conf` `file` with `state: absent`

The following diagram depicts the required nginx configuration:



Setting up a load-balancer/reverse proxy requires the following configuration (below is just an random example)

A configuration file for the upstream (part of http directive) name for example `conf.d/example-lb`

```
upstream example-servers {
    server 192.168.1.10;
    server 192.168.2.201;
    server 192.168.2.202;
}
```

And then a server

```
server {

   listen 7070;

   location / {
       proxy_pass http://example-servers;
       proxy_set_header        X-Real-IP          $remote_addr;
       proxy_set_header        X-Forwarded-Port   ;
       proxy_set_header        X-Forwarded-Host   $host:;
       proxy_set_header        X-Forwarded-Server $host;
       proxy_set_header        X-Forwarded-Proto  http;
   }

   error_page   500 502 503 504  /50x.html;
     location = /50x.html {
       root   /usr/share/nginx/html;
   }
}
```

We will guide you through the steps. Our configuration will be very similar to the one above, in conjunction with the following information:

port	upstream	conf file (http)	conf file (server)
80	front-end	`conf.d/loadbalancer-front-end`	`conf.d/server-front-end.conf`
9090	rest-servers	`conf.d/loadbalancer-rest-servers`	`conf.d/server-rest-servers.conf`

As you can see from the table, for each upstream you;ll need to create two file. For both you will create a template. You will then use a loop to create the two files *front-end* and for *rest-servers*. This however means you want to loop over multiple tasks. This is only possible using a `include_tasks`

Loop over multiple tasks

- Add an `include_tasks` to a new file to be created by you named `includes/conf-nginx-lb.yaml` (we work on its contents further below)
- What is the looping data you need? The only variable pieces of data are the port and the name of the inventory group. Therefore create a loop for the following two dictionaries `loop:` :

```
- {upstream: front-end,  port: 80}
- {upstream: rest-servers,  port: 9090}
```

- make sure you notify to reload nginx (add also the handler as you've done before)

Create a new file named `includes/conf-nginx-lb.yaml` to hold the tasks over which you want to loop:

- First "rename" the looping variable name to `proxy` by setting a fact `proxy: "{{ item }}"`
- Then create a task that takes the template file named `templates/upstream.conf.j2` and produce `/etc/nginx/conf.d/loadbalancer-….conf` (replace the three dots with the name of this `upstream` from your looping variable `{{proxy.upstream}}`). Do you recall the name of the module `template`
- Create a second task that takes `templates/server.conf.j2` and produces `/etc/nginx/conf.d/server-….conf`

Template files

Let's create the template files. Start with `templates/upstream.conf.js` :

- Specify the name of this upstream using your looping variable `{{proxy.upstrea}}`
- You will then have to iterate over each host in the group with that name `{% for host in groups[ proxy.upstream ] %}` and
  - get the ip4 address using the host var's `hostvars[host]` attribute `['ansible_eth1']['ipv4']['address']`)
  - don't forget to close your loop `{% endfor %}`

Then open `templates/server.conf.j2` :

- Specify the listen port using your looping variable `{{proxy.port}}`
- And the set the `proxy_pass` value to your upstream url `http://{{proxy.upstream}}`

Change the rest-servers

Did you notice the `X-Forwarded-…` header values? These are passed on to the proxy targets. It is especially

important for the rest-service to use these values and not the server/port it is really running on (which would be `localhost:80`). Dynamic web application (like the swagger-ui) and rest services use often the server name and port to construct URLs. it is therefore important it uses the values of the proxy server.

You've already set the headers. You will just need to configure the application and set the environment variable `server.use-forward-headers`. So go ahead and add this environment variable to `~/course/ansible/worldstack/rest-server.yaml`

Test

Rerun the playbook for the rest-server and run your load-balancer playbook.

You should now be able to access the rest server using the load-balancer: swagger on http://10.20.1.2:9090/swagger-ui.html (http://10.20.1.2:9090/swagger-ui.html) or to test the service or use cURL again:

```
$ curl http://10.20.1.2:9090/countries?continent=africa | jq .
```

If you still have the front-end deployed you should be able to go to http://10.20.1.2 (http://10.20.1.2) and see a list of countries.

Wow, everything is working (well it should anyway). If not try to fix your problems. Remember the troubleshooting techniques your learned earlier in the course.

Let's improve the front-end play in the next step.

Everything should be working. There is still a few things we will do to improve things. Let's copy the `~/course/ansible/front-end/frontend.yaml` over to the `worldstack` directory.

First change the reference to `index.html` and `front-end-opt.js`. These are now available in the `files` sub-directory.

Now did you notice you have two playbooks with similar tasks. Both this one and the `load-balancer.yaml` install nginx (add the yum package and install the packages). This is not very DRY (https://en.wikipedia.org/wiki/Don%27t_repeat_yourself) and we should define these tasks once and then just include them when we need them.

So go ahead and create a file named `includes/install-nginx.yaml` and then move two *two* installation tasks (`yum_repository` and `package`) to this file. Once you have done that, replace the tasks inside `frontend.yaml` and `load-balancer.yaml` with an include `include_tasks`

Test your updated playbooks

Then check if you still get a list of countries at http://10.20.1.2 (http://10.20.1.2)

Now that you have all the playbooks in place it would be nice to create one master `main.yaml` that includes all your playbooks.

Create a file named `main.yaml` and add four includes to your playbooks `import_playbook`

To test, reset all your machines to their initial state (run this from the `machines` directory)

```
$ vagrant snapshot restore initial
```

And then run your main playbook and see how it populates your infrastructure.

Test by going to http://10.20.1.2 (http://10.20.1.2)