Lab 4

Playbook handlers

You've just completed the lab in which you created a playbook to setup the database. In this lab you will be setting up the front-end. The result might not be as satisfying because we don't have the required rest-services in place yet. So the result will be a page with an error that the service can't found. But that page will already be the proof that you've deployed the front-end app successfully. The front-end will be served from a nginx server. You will need to provision the server, configure it and then deploy our application.

The first thing you want to do is to ensure your target machine is reset to it's initial state in case you have placed with it during demos or previous labs.

Open a terminal to `~/course/ansible/machines` and runs

```
$ vagrant snapshot restore machine-3 initial
```

This front-end is a simple application using javascript to communicate with your restful endpoint. The application was build using Scala.js (https://www.scala-js.org) and Binding.scala (https://github.com/ThoughtWorksInc/Binding.scala). This might not be very relevant, but it means you will have to build the application.

To compile and build you'll need an environment with Java, Scala and SBT. To make things easier we will ask you to build the application using a docker container with such an environment.

Open a terminal and navigate to `~/course/ansible/front-end/project` . Then compile the project using a scala-sbt (https://hub.docker.com/r/hseeberger/scala-sbt/) build container:

```
$ docker run --mount type=bind,source="$PWD",target=/root --rm \
    hseeberger/scala-sbt:8u181_2.12.6_1.2.1 \
       sbt fullOptJS
```

This will produce a compiled Javascript file `target/scala-2.12/front-end-opt.js` . Later during this lab, you will make sure this will be available to the nginx server.

Navigate to `~/course/ansible/front-end` (one directory up). Like before you will find the following files:

- An `ansible.cfg` with some default settings
- An almost empty `frontend.yaml` for you to complete (we have provided the preamble)

We will leave you a little more on your own, as many of these steps you have already performed in the previous lab.

Install and run nginx

You will need to install nginx-1.14.0, which is not available in the default centOS repository. We will therefore need to register a yum repository. Check the module index (https://docs.ansible.com/ansible/latest/modules/list_of_packaging_modules.html#os-packaging-modules) for a correct module `yum_repository` . We are using the following values for it (originates from nginx docs (https://www.nginx.com/resources/wiki/start/topics/tutorials/install/#official-red-hat-centos-packages)):

- 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` (you will need to set this value)

Then Install the following packages:

- `ca-certificates`
- `nginx-1.14.0`
- and make sure the `nginx` service is enabled *and* started

Install the application sources

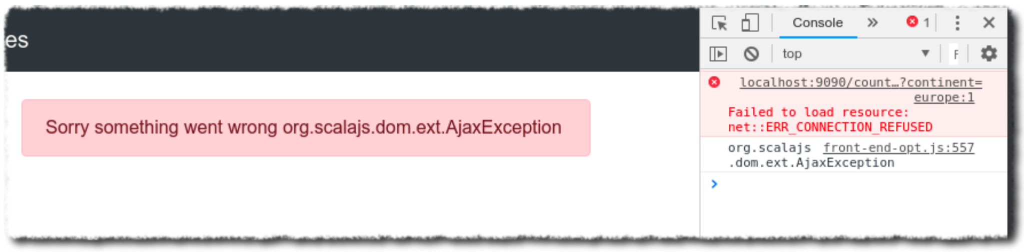
Copy the following files:

source	destination
`project/index.html`	`/usr/share/nginx/html`
`project/target/scala-2.12/front-end-opt.js`	`/usr/share/nginx/html/js/`

Run the playbook

Go ahead and run the playbook.

After running the playbook successfully you should be able to go to the web application using the url http://10.20.1.3/ (http://10.20.1.3/). As we mentioned you will see an error:



Sorry something went wrong org.scalajs.dom.ext.AjaxException

```
localhost:9090/count…?continent=europe:1
Failed to load resource: net::ERR_CONNECTION_REFUSED
org.scalajs.dom.ext.AjaxException    front-end-opt.js:557
```

But this is proof nginx is running and the application has been successfully deployed!

In the next step you will configure the nginx server a little further.

You will do some basic configuration of the nginx server. Not all the configuration has easy to observe results. Some of them do, so we can use those to ensure your configuration is working.

One thing you should now about the default nginx configuration on centos is that there is a `/etc/nginx/conf.d` directory for custom configuration. The standard `/etc/nginx/nginx.conf` includes any `*.conf` available in that directory:

```
http {
    include       /etc/nginx/mime.types;
    …
    include /etc/nginx/conf.d/*.conf;
}
```

Notice how `conf.d/*.conf` files are applied to the `http` block. One file named `default.conf` is already present inside the `conf.d` directory and it configures a basic *server* on port 80.

You will add `conf.d/*.conf` files and change the `default.conf`

But before that, let's have a look at the current behaviour of the nginx server.

Starting situation

Check the headers for your page using cURL:

```
$  curl -I http://10.20.1.3/index.html
HTTP/1.1 200 OK
Server: nginx/1.14.0
Date: Mon, 13 May 2018 11:51:00 GMT
Content-Type: text/html
Content-Length: 925
Last-Modified: Mon, 13 May 2018 12:02:09 GMT
Connection: keep-alive
ETag: "5af82941-39d"
Accept-Ranges: bytes
```

Note the following:

- `Server: nginx/1.14.0`
- There are no `Expires` or `Cache-Control` headers

"harden" security

You will now change your existing playbook. We urge you not to run your playbook as you will need to register a handler in the next step.

We will add some basic "hardened" security to nginx. You will add some configuration to the `http` configuration. This can be achieved by ensuring a new file named `/etc/nginx/conf.d/security.conf` . Make sure this file has the following contents e.g, using `blockinfile` :

```
server_tokens off;
client_header_buffer_size   4k;
large_client_header_buffers 8 8k;
client_max_body_size 20m;
connection_pool_size 8192;
request_pool_size 8k;
```

From all the configuration, make a special note of `server_tokens off` . We will remind you later.

Caching

For caching we will need to add two sets of directives.

First a map between mime-types and expiration periods. This needs to be applied to the `http` section. Therefore make sure the following contents is available in a file named `/etc/nginx/conf.d/caching.conf` :

```
map $sent_http_content_type $expires {
    default                    off;
    text/html                  epoch;
    text/css                   max;
    application/javascript     max;
    ~image/                    max;
}
```

Then use the `$expires` map inside the existing `server` configuration by ensuring the following contents to

`/etc/nginx/conf.d/default.conf` (add it before the line `#error_page 404`) you can use `lineinfile` here:

```
expires $expires;
```

Hopefully you have *not* ran the playbook and please *don't* run it now either. You'll still need to make sure the server configuration is reloaded. You will do that using a handler in the next task.

You are changing the configuration of nginx three times. *Each* of these would require a service reload. However one time is enough. This is a perfect example case for handlers.

Handler

Let's start by defining the handler.

- Add the correct section `handlers:` (sibling of `tasks`)
- Add a single handler named *reload nginx* `- name: reload nginx`
- The handler should reload the nginx service `service` module with `state: reloaded`

Trigger the handler

Now for each of your "configuration" tasks ("hardened security", "caching map" and "applying it to the server"):

- trigger the handler `notify: reload nginx`

Now rerun your playbook. Notice it runs your tasks, and then runs you handler at the end.

Test results

To test that the configuration now has been applied, run your cURL command again:

```
$ curl -I http://10.20.1.3/index.html
```

This time, notice:

- The version no longer shows (`Server: nginx`). This is due to `server_tokens off`
- There are `Expires` and `Cache-Control` headers in place now

Instead of triggering your handler by name, have it listen to a trigger name `listen: …` .