CLOUD COMPUTING – CLASS PROJECT

Project by:

**Pradyumna S – 01FB14ECS151**

**Shreyas S Rao – 01FB14ECS231**

## **Topic**: Creating a Platform-as-a-Service on the cloud

## **Abstract:**

The main objective of this project is to create a PaaS on the cloud so that any given user can sign up for our service and use our service (platform) to launch their own product (website).

We have made our service as **generic** as possible, so that not only can a tenant upload his own custom applications, but, we as the cloud provider, can launch our cloud service on any given server (bare server), provided we have its IP address and login credentials.

Our cloud service also provides support for multi-tenancy, so that multiple websites can be launched and maintained on the same platform. This enables our platform to be launched at an enterprise level on the market, commercially, if necessary.

## **Technologies used:**

* **Linode Server**: We are currently using an online service to rent out a virtual server. We decided to do this, keeping in mind the amount of load our cloud service will bear on the server it’s running on, and a VM running on a laptop wouldn’t be sufficient for the purpose. Also, by using an online server, we are able to host our service on the internet, so that it can be used from anywhere in the world.
* **Docker containers**: Each component of our service runs on a separate container. The main advantage of using containers over VMs is that unlike VMs, they consume as minimum resources as possible, thus, hundreds of containers can be started on a single server, which all run at bare metal speeds. Each container runs on an individual port on the server.
* **Nginx**: This is our reverse proxy service. Running this on the server allows it to be accessible on the internet, ie, any application running on the localhost of the server is available on its IP address on the internet, and hence, our service becomes “on-line”.
* **Flask on Python**: All the containers running on the server (other than the DB container, which will be explained in the next section) run Flask applications, and for the moment, even the tenant’s applications are required to be Flask applications. The reason for using Flask is simple – fast development cycle and well-defined programming paradigm. Plus, we were already familiar with Flask as our first two Cloud Computing assignments were Flask applications.
* **Python scripts and shell scripts**: We need to use dirty scripts to enable **AUTOMATION**. Effectively, once we have a bare server with Ubuntu 16.04 LTS installed, along with its IP address and root login password, all we need to do is remotely run **one single command to setup and deploy the entire cloud platform on the server**.

## **Design:**

Consider the following roles that we’ll refer to frequently in our discussion:

* **Us**: the cloud providers, i.e., us. We launch and host the PaaS for users to host their products.
* **Tenants**: the users of our PaaS. They upload their web application to our server in order to have them hosted online.
* **End-users**: the users of the tenants. They are the normal users of the web applications belonging to the tenants, but hosted by us (our cloud platform).

We (our script) start the following Docker containers on the server:

* **Host container (port 8080)**: It is the “main server” that runs a Flask app that acts as the gateway for communication between any user on the internet (tenant or end-user) and our microservices. It differentiates between a tenant request and an end-user request (by validating the token sent by the user) and takes an appropriate action. The tenant controls are hosted on this container, which allow a tenant to sign up for our service, login, upload their web-role and worker-role web apps, and run queries on the database. All end-user requests are load-balanced and forwarded to the selected web-role container.
* **Token generating service (port 9001)**: During tenant signup, a corresponding user is created on the database, with appropriate privileges, so that all DB queries from that tenant’s side are run as the corresponding DB user. When a tenant logs into our tenant-controls page, their credentials are validated against the database, and a corresponding “token” is generated for them, which is saved on their system locally. Henceforth, until they logout, all their requests to the service will also contain their token, so that the other microservices need not repeatedly consult the database or the TGS for each and every request from tenants. Thus, the TGS provides **one-time authentication**.
* **Load balancer (port 9002)**: a stand-alone microservice that load-balances each tenant’s end-users’ requests and forwards them to the assigned web-role container. It receives a request from the “host” that contains details about which tenant’s web app is to be load-balanced, and a list containing the IP addresses (along with port numbers) of that tenant’s web-role containers, and based on the “liveness” of each container (whether the container is currently active) the current load on each container, the one with the least load is selected and returned to the host. We have designed this in such a way that **new web-role containers can be added on-the-go, and it also accounts for inactive web-role containers**.
* **Database (port 3306)**: Is a Docker MySQL container that runs a database and receives MySQL queries and processes them and returns results to the service that made the query.
* **Conductor (port 9003)**: This is the interface to the database. As no tenant should be allowed to make direct calls to the database, an interface such as this one, pre-processes the query and makes sure it is safe before forwarding it to the database itself. The conductor does the following functions:
* Validates the request based on the token in the request
* Sanitizes the query
* Converts the tenant’s raw query into a query that adheres to the shared table’s columns, in order to support multi-tenancy.
* Forwards the query to the database
* Receives results from the DB and returns appropriate result to the caller, i.e., it ensures that MySQL errors aren’t returned to the caller, hence **preventing SQL Injection**
* **Web-role containers (ports 9100-9199)**: These are created when a tenant uploads their web app (a Flask application in a ZIP file format). The tenant can also specify how many web-role instances to create. These containers host the tenant’s web app, and end-user requests are forwarded to these. Web role does the job of serving the end-user with a front-end website, and receives requests and takes appropriate actions, like updating the database, etc.
* **Worker-role container (port 9200)**: Again, this is created when tenant uploads their program. Only one worker role per tenant can be run. The program uploaded is a normal Python program that has a “while True” loop so that it perpetually runs on the server as a background process. Worker role usually does analytics jobs or data-fetching jobs in the background, and updates the database accordingly.

## **Workflow:**

### **For creation of the cloud service**: (assuming we have all the files required for launching the service, and the IP address of the server and its root login password) – **all of the following steps are achieved using a single command:**

* Send all the necessary files to the server using “scp”
* “ssh” into the server and do the following:
  + Reconfigure ssh
  + Install and configure Nginx
  + Install and configure Docker
  + Create MySQL database container
  + On the MySQL server, create database “tenants’ that must be used by tenants
  + Create conductor container
  + Create token-granting-service container
  + Create load-balancer container
  + Create host container

### **For tenant**: (assuming that the tenant has their Flask web app in a ZIP file)

* On their browser, access http://<server\_ip\_address>/tenant/
* Sign up
* Log in
* Select zip file to upload for web-role, and number of instances of web-role to run. The app routes specified in the Flask web app should preferably have the prefix “/<tenant\_name>/” for the purpose of multi-tenancy
* Select zip file to upload for worker-role
* Select SQL file to run on the database, to initialize (create and populate) the tables the tenant will be using.
* Run individual queries on the database and see results

### **For end-user**: (assuming that the tenant has already hosted their web app on the service, as, if they haven’t, a 404 error is returned to the user)

* Access any tenant’s website by typing into the browser http://<ip\_address>/<tenant\_name>/<app\_route\_defined\_by\_tenant>
* The remaining workflow for the end-user is defined by the tenant in their Flask application.

## **Conclusion:**

“Cloud computing” is mostly about making the internet more friendly to use, both for end-users, as well as those who wish to use virtual resources to achieve their own tasks like hosting websites online or to simply run resource-hungry applications which they cannot run on their local machines, or for enterprises that wish to reduce their reliance on physical servers that cost a lot to setup and maintain.

In such an era of “virtualization”, cloud-hosting platforms, like our project, are in high demand, primarily due to the low amount of effort needed by the tenant to host his/her website online. And the high amount of automation that we have incorporated into the project makes it very easy to deploy the service itself on any given server without much hassle, in addition to tenants being able to easily host their websites, and allows us to monitor them as well.