**Lab 4: Cloud Platform**

In this lab, we’ll be leveraging various services from Amazon Web Service (AWS) and other popular DevOps tools to deploy a representative web application on cloud.

**Disclaimer:** This lab might cost your money. Please keep track of free tier limit before you start any service. FYI, if you have never applied before, GitHub student developer pack will give you [$150 credit on AWS](https://education.github.com/pack/offers#aws-educate).

**0. Background**

You just started your awesome startup CloudNet and you quickly put together the very first version of your website. You had one lab machine connects to the public internet, registered a domain name that resolves to it, and hosted your web application there. You typed the last command needed for running the server, and told every friend to visit your awesome website. In the next few days, people started registering: 10, 100, 200… They give compliments on how great the site is, and you are getting more users. It’s been great.

However, with increasing volume of traffic, you started getting complaints saying that your website is really slow. Analysis of your operation log shows that not only your website have 100x more visitors than last time you check, but these visitors come from different part of the world as well. You started thinking seriously about how to scale up your website, and how to deploy it on more than one machine. You heard the buzzword “cloud” and how great it is all over the media, and you decided to try it out. After some researching and comparing, you decided to choose AWS. And here starts our journey.

**1. Getting Started**

We supplied you with a Django web application “CloudNet”, go to your repository and pull the source code. We’ll first try running the application locally. This section is for you to get familiar with the overall setup, and there’s nothing you need to submit or include in your report for this section.

Note: You will be modifying some of the Django code in this lab, but it’s not part of this lab’s (or even this class’s) requirement that you become a [Django](https://www.djangoproject.com/) or [ReactJS](https://reactjs.org/) expert. We do encourage you take a quick look at online resources for these two frameworks though, as both of them are pretty popular, and you need to have some basic understanding of a typical frontend-backend web application in order for this whole lab to make sense.

After pulling the source code, install necessary dependencies: (Assuming you have python, pip, node, and npm installed correctly. Use Google for troubleshooting.)

|  |
| --- |
| $ cd cloud\_net  $ pip3 install -r requirements.txt $ npm install  $ npm run build |

Note: It’s recommended (but not required) that you use Python [virtualenv](http://docs.python-guide.org/en/latest/dev/virtualenvs/) to manage Python-related dependencies.

After these dependencies are installed, run the web application by:

|  |
| --- |
| $ npm start |

Open a web browser and go to <http://localhost:8000/frinet/>. You should see the website, and you should be able to register, login, post a message, and make friend with others (you need to know others username for this).

Navigate through the website and try different functionalities. This website is a minimal representation of nowadays fancy social websites: the functionalities are limited, but the frameworks used is industry standard, and is realistic enough to be extended to a way more complicated application.

Awesome, this concludes your local setup. In the real world, everything starts from local development and testing. At this point, you are ready to try deploying your application on the cloud, along with more complicated setup.

**2. First Taste of AWS**

Let’s start by registering an AWS account, and creating 1 EC2 instance (a VM). Make sure you choose the ones that are under the free tier you will be limited to 750 hours free per month. Feel free to just leave the image configuration as default. After you launch the instance, make sure you can SSH into it.

**2.1 Database with MySQL**

Previously when you test the web application locally, you are using Django’s default backend database [SQLite](https://www.sqlite.org/) (via sqlite3 interface). This is great for local development and testing, but owing to its serverless nature, neither does it support multi-user applications, nor does it scales very well. Thus we won’t use it for production databases. Instead, we’ll replace our production backend database with [MySQL](https://www.mysql.com/). Django comes with native MySQL support. See [this page](https://docs.djangoproject.com/en/2.0/intro/tutorial02/) for more information.

Task:

- Install MySQL on the EC2 instance you just created. Change the backend database to MySQL, create a user for your MySQL database with username “admin”, and a password of your choice. Make sure the application works as before with your new database setting.

- In a few sentence or using a table (include in your report), compare and explain the major difference between MySQL and SQLite.

**2.2 Caching with Memcached**

As you might know, performing database queries are expensive. In order to minimize request latency, a in-memory caching layer is normally used in between database and web application for caching query results. We will be using [Memcached](https://memcached.org/) for this purpose. It’s an in-memory key-value store for small chunks of arbitrary data.

Django has [integrated support](https://docs.djangoproject.com/en/2.0/topics/cache/#memcached) for Memcached. You will need the python-memcached module for using the API. Installing Memcached itself depends on the operating system you run on.

Task:

- Modifying the application so that it uses Memcached for users’ ‘Message’ content caching. (Modify ./cloud\_net/settings.py and ./frinet/view.py file to make use of Django Memcached api)

- In a few sentence or using a table (include in your report), compare Memcached with another popular caching framework [Redis](https://redis.io/), explain why you might want to use one over the other.

- **(Extra Credit)** Try doing caching using Redis. If you choose to do this, you still need to finish the Memcached setup, but make a separate branch with Redis settings and push them to your repo.

**2.3 Web Server with Nginx**

Django comes with a lightweight web server, and that’s what we’ve been using until now (when you do **npm start**). But this embedded web server is only intended for local development and testing purpose. In fact, you should never use this in production environment as pointed out by Django’s official website [here](https://docs.djangoproject.com/en/2.0/ref/django-admin/#runserver). So we need to replace the web server with a more “realistic” one.

For this purpose, we will be using [Nginx](https://www.nginx.com/), one of the most popular web server frameworks.

However, there’s another piece missing. A web server faces the outside world. It can serve files (HTML, images, CSS, etc) directly from the file system, but it can’t talk directly to Django applications; it needs something that will run the application, feed it requests from web clients (such as browsers) and return responses.

A Web Server Gateway Interface - WSGI - does this job. WSGI is a Python standard. uWSGI is a WSGI implementation, and this is what we will be using. Here’s a nice [tutorial](http://uwsgi-docs.readthedocs.io/en/latest/tutorials/Django_and_nginx.html) for how to all these together. (Note: if you run into issues with uWSGI using the wrong python version, try building uWSGI from source as instructed [here](http://uwsgi-docs.readthedocs.io/en/latest/WSGIquickstart.html), and explicitly set which python version you want uWSGI to build with. e.g. run python3.5 uwsgiconfig.py --build before running make, and make sure you are running the correct uWSGI binary when you make the change.)

Task:

- Replace the default Django web server with Nginx. Make sure your web application still works as before.

- In a few sentence or using a table (include in your report), compare Nginx with [Apache Web Server](https://httpd.apache.org/), explain why you might want to use one over the over.

**2.4 Docker**

Now you are ready for production deployment. Production environments vary, and your application have complex dependencies - both made your deployment experience unhappy. Fortunately, recent advances in container technique makes software deployment much easier. A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings. Containerized software will always run the same, regardless of the environment.

Docker is a containerization tool of such. The philosophy is “develop, deploy, and run”. Docker make deploying application fast and safe (it wouldn’t mess up the machine). It is like a virtual machine, but share kernel with host machine. Therefore it is faster and lighter than virtual machine.

Dockerfile is the script file where Docker read at building and starting up a new container.

|  |
| --- |
| docker -f /PATH/TO/DOCKERFILE <context path> |

Here are some simple explanation of Dockerfile syntax:

|  |
| --- |
| RUN: RUN <command> or RUN ["executable", "param1", "param2"]  (as you use shell)  FROM: FROM <image>[:<tag>] [AS <name>]  ENV: ENV <key> <value> (define environment variable) |

Don’t use “RUN” for each command because it will make your docker image extremely large. Instead, “&&” is more useful here. To further reduce the image size, remove redundant files after build.

Don’t include “node\_module” directory in your image (it is pretty big). Instead, use “npm install” to install node dependency on target machine.

For more advance syntax, please refer to [Dockerfile reference](https://docs.docker.com/engine/reference/builder/#usage).

Task:

- Build a docker image of the web application, run it locally on your EC2 machine. Make sure your web application still works as before.

(Note: You might want to switch back to sqlite3 when building the docker image, since building MySQL within docker hasn’t been fully tested by the course staff yet.)

**3. Scale up with AWS**

In section 2, you successfully setup a web application that runs “in cloud” on one machine. As you could imagine, this setup does not scale very well. Just to be clear, writing scalable application is itself an interesting yet challenging topic that could take a whole semester to discuss, and it’s not our main goal here. We want to provide you with a better sense of how to leverage cloud services (like those provided by AWS) to deploy scalable applications.

There are different ways to leveraging these cloud services. If you stick with the same methodology as in section 2 -- that is, only use EC2 instances as the infrastructure and have complete freedom in choosing application platforms (also known as IaaS, or Infrastructure as a Service) -- you will probably have a bunch of EC2 instances running different services. For example, you could imagine having a couple instances running Django + ReactJS code, a web server, and probably a load balancer, having another couple instances running mysql, having another couple instances running Memcached, etc. And you need to configure each instance in a similar way as you did in section 2. There are tools to automate these configuration (e.g. [Ansible](https://www.ansible.com/)), but this is not very satisfying when you just need to run a typical web application and you don’t want to deal with all the messy setup or subtle resource allocation.

Fortunately, you don’t have to do all these. AWS provides another set of service that allow you to just upload and run your application on, without the complexity of managing the underlying infrastructure typically associated with launching the application. These are known as PaaS (Platform as a Service), and we will be experiencing some (RDS, ELB, EBS, and ElastiCache) in this section.

**3.1 Setup Virtual Private Cloud (VPC)**

Amazon Virtual Private Cloud (Amazon VPC) lets you provision a logically isolated section of the AWS Cloud where you can launch AWS resources in a virtual network that you define. Your VPC closely resembles a traditional network, with the benefits of using AWS's scalable infrastructure. You have complete control over your VPC; you can select the IP address range, create subnets, and configure routes and network gateways. You’ve been using VPC in section 2 -- when you launch a EC2 instance, you could assign a pre-configured VPC to it, or AWS will create a default VPC for that instance.

Although default VPCs is generally good enough when you use AWS services, let’s try creating our own VPC to have a better sense of what the components are and what services it provide. Later in this section, we will be launching different services in the VPC you create.

Ideally, you should have multiples VPCs for different purposes. (eg. One for staging, and another one for production. The idea behind the staging area is to have a ”practice” area to deploy new services. Many enterprises will upload early versions of their service and test them out in the staging area, hammering them with fake traffic or directing a small number of users there for a test. ) For simplicity, we’ll only setup one production VPC.

[**CHALLENGE TASK:** *This paragraph can be done in lab3 groups, and also it's extra credit - note staff may not have cycles to help with this particular step:*] Try connect your cloud to your on-prem network (the network you set up in lab 3) via a VPN connection. Terminate the connection on your prem side in your ASA router. To do this, you'll need to replace the Internet machine you used in lab 3 with a default route pointing to the wall jack CITES runs. Use DHCP to acquire an IP address for the Internet-facing port on your ASA. Note AWS will help you generate a configuration for your ASA router. Finally, for even more extra credit, set up VLAN 4 in your lab3 topology to reach subnets 10.1.2.0/24 and 10.1.3.0/24 in your lab3 network.

Task:

- Create a VPC with prefix 10.1.0.0/16. Create 3 subnet within this VPC with prefix 10.1.1.0/24, 10.1.2.0/24, and 10.1.3.0/24. The first one is intended for web subnet, and the latter two are intended for database subnet.

- Create two [security groups](https://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/VPC_SecurityGroups.html) and attach them to your VPC: one for your web server that allows public access on HTTP, and one for the database instances (see below) that only allow traffic from your web server. (A reference setup is provided [here](https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Tutorials.WebServerDB.CreateVPC.html).)

- Create a [Internet Gateway](https://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/VPC_Internet_Gateway.html) and attach it to your VPC.

- **(Extra Credit)** Set up VPN connection with the setup you did in lab3.

**3.2 Setup Relational Database Service (RDS)**

Now that we have the VPC, let’s deploy our service, starting again from the database. We’ll be using Amazon Relational Database Service (Amazon RDS), which makes it easy to set up, operate, and scale relational databases in the cloud. You have the option to use different database engines for RDS, and let’s stick with MySQL, which is RDS Free Tier eligible.

For more information regarding launch RDS instance in VPC, see [here](https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_VPC.WorkingWithRDSInstanceinaVPC.html).

Task:

- Create a RDS MySQL instance as instructed [here](https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide//USER_CreateInstance.html).

- Modify your Django settings to use RDS, as instructed [here](https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create-deploy-python-rds.html).

**3.3 Setup ElastiCache**

Let’s replace our local Memcached server with the caching cluster AWS provides, and we will be using AWS ELastiCache for this purpose. ElastiCache is a web service that makes it easy to set up, manage, and scale distributed in-memory cache environments in the cloud. It provides a high performance, resizable, and cost-effective in-memory cache, while removing the complexity associated with deploying and managing a distributed cache environment. Amazon ElastiCache is protocol-compliant with Memcached, so the code, applications, and most popular tools that you use today with your existing Memcached environments will work seamlessly with the service.

Task:

- Create a ElastiCache cluster for Memcached, as instructed [here](https://docs.aws.amazon.com/AmazonElastiCache/latest/UserGuide/Clusters.Create.CON.Memcached.html).

- Modify your Django setting so that it uses ElastiCache as backend caching. You will need to install django-elasticache module as instructed [here](https://github.com/gusdan/django-elasticache).

**3.4 Deploy Application with Elastic Beanstalk (EB)**

AWS Elastic Beanstalk is the fastest and simplest way to get an application up and running on AWS. You can simply upload your application code and the service automatically handles all the details such as resource provisioning, load balancing, and auto-scaling.

In this part, we will be using EB with Docker. You have the Docker setup in Section 2.4, and you had it running locally. Now we ask EB to run it for us. Note that you might want to rebuild your Docker image since you modified your source code in 3.2. When you deploy your application, Elastic Beanstalk provisions one or more AWS resources, such as Amazon EC2 instances. The software stack that runs on your Amazon EC2 instances depends on the container type. If you want better control, you could SSH into the EC2 instances created by EB just like you did in Section 2.

Task:

- Create a new EB application. Configure the environment of this application to be single instance with Docker, and place the instance into the web subnet you created in 3.1.

- **(Extra Credit)** Configure the environment to be using Load Balancer and AutoScaling instead of single instance. Make a note in your report if you do this. Write a paragraph about what you observe with using auto scaling, and what are the benefits.

- Modify Environment Properties to attach the RDS you created in 3.2 to this EB application, as instructed [here](https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/AWSHowTo.RDS.html#rds-external-defaultvpc).

- Modify your ElastiCache security group to allow access from EC2 security group used by your EB application for proper caching, as instructed [here](https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/AWSHowTo.ElastiCache.html).

At this point, your CloudNet application are fully deployed on AWS. You should be able to verify that it’s been deployed correctly by visiting the public ip of your EB’s EC2 instance (or the public ip of your ELB if you choose to do extra credit ELB with auto scaling).

**4. AWS API**

You’ve been using AWS Console a lot. That’s a nice GUI indeed, but sometimes we want to do things in a more programatic way.

AWS offers multiple language API to manipulate and retrieve information of your VPC, EC2 instances, etc. For more information, please reference AWS official [documentation](https://aws.amazon.com/tools/?nc1=h_ls).

If you are using Python AWS API, install by running

|  |
| --- |
| pip install boto3 |

Remember to include you credential ID and key follow this [tutorial](https://boto3.readthedocs.io/en/latest/guide/quickstart.html). The ID and key can be found in IAM (Identity and Access Management) of AWS.

Task

- Pick your favorite language, write a program that uses AWS API to print out information regarding your previous setup in section 3

i.e. As a general guideline, print your RDS instance information, EB instance information, ElastiCache information, Elastic LB, VPC.

- **(Extra Credit)** Do section 3 using AWS API instead of the GUI Console. Include your code in your submission.