Deployment and Server Management

Fall 2018 – Introduction to Full-Stack Development

Instructor: Matthew Fritter

matt@frit.me

Stepping Back: Flask

- Last week, we discussed some additional functionalities in the Flask framework:
  - Handling static pages such as JS and CSS resources
  - Redirects
  - Sessions
  - Template Inheritance
  - Discussion of databases
- After completing lab 8, you should be at least acquainted with building web applications in Flask

Stepping Back: Git Deployment

- Last week we also discussed using Git for deployment:
  - Reviewing the workflow of local commits, pushing to a remote repository, pulling from a remote repository
  - Ideally, you have a dev branch that is used for code in-development, and a production branch that is for clean, tested code
    - Keep development code away from production code!
  - Use bash scripting to set up your environment quickly, and set up automatic pulls from the production branch to keep our web server up-to-date

Stepping Back: Git Deployment

- We also touched briefly on the shortcomings of using Git as a deployment solution:
  - Environment setup scripts will be system-dependent – you'll have to make sure your initial server configuration is the same each time
  - Bash scripts can be susceptible to changes in dependencies and other unforeseen issues – for example, a package might be moved to a different repo, or a web resource that you fetch might disappear

Announcements

- Quizzes 1 and 2 are marked and can be picked up – the marks should be uploaded to Moodle – Overall average is relatively high
- Marks for labs should begin to be posted this week
- The lab 8 due date has been extended until next Wednesday (Nov 21st), but there will still be a lab 9 tomorrow
- Are there any questions on what we covered in Flask, on Git deployment, or regarding the quiz?

Flask Quiz

- This quiz will test you on what we covered in Flask from the lectures and the labs
  - There will be some code writing and some conceptual questions
- This quiz is closed book – please close/put away laptops and phones, and refrain from talking to neighbours

Deployment with Docker

★ Containerization

- Docker is a system for deploying applications using *containers*.
- Containerization is a type of OS-level virtualization
  - The OS allows multiple individual user instances to exist
  - The instances share the OS kernel, but are otherwise isolated from each other and operate as individual machines as the application level
- The basic concept: pack everything we need for our web application into a container, and then that container can be cloned onto other machines and run as an individual instance

Is this a Virtual Machine?

- While containerization might look similar to virtual machine systems like VirtualBox or Vmware, it’s actually different
- A Virtual Machine contains a full operating system and accesses computer resources via a hypervisor
  - VirtualBox and Vmware use what are known as hosted hypervisors – any request for computer resources must run from the VM through the hypervisor to the host OS, which must manage those requests
  - Depending on the hardware specifications and host OS, this can be very slow compared to a native OS
- In comparison, the container doesn’t contain an operating system – it shares the underlying native OS with other containers.

★ What Does Docker Do?

- Docker is a utility designed to manage, build, and run containers
  - Supported on Windows, Mac, CentOS, Debian, Fedora, and Ubuntu
- It provides a command line interface to handle containers and run commands within a container
- It also hosts a registry, similar to a package manager repository or GitHub, which allows you to fetch containers from the web
- Note that containers are still OS-specific – you’ll run into problems if you try to run a CentOS container on a Windows machine, for instance

★ Installing Docker

- Luckily, Docker is fairly easy to install. Simply run the following commands:

```
sudo yum install -y yum-utils device-mapper-persistent-data lvm2

sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo

sudo yum install docker-ce

sudo systemctl start docker

sudo systemctl enable docker
```

- You can test your install by running:

```
sudo docker run hello-world
```

  - This should download the hello-world container from the Docker registry and run it, generating a hello message

★ Installing Docker

- Luckily, Docker is extremely easy to install. Simply run the following commands:

```
sudo yum install docker-ce
sudo systemctl start docker
sudo systemctl enable docker
```

- You can test your install by running:

```
sudo docker run hello-world
```

  - This should download the hello-world container from the Docker registry and run it, generating a hello message

★ Preparing Docker for CentOS

- Now that we've installed and verified that Docker is working properly, we can go ahead and download the CentOS image for Docker:

```
sudo docker pull centos
```

- Think of this as the baseline image for running a Docker container on CentOS. When we develop our own images, we'll develop them off of the CentOS image
  - You can view which images you have installed/pulled to your machine using:

```
sudo docker images
```

★ Interacting with a Container

- Now that we've pulled the CentOS image, we can run it as a container. We use additional flags to tell Docker that we want to interact with the container via the command line:

```
sudo docker run –it centos
```

- After running this command, you'll notice that we're now the root user, and we are no longer operating on the server, but some garbled alphanumeric string
  - The alphanumeric string is the container ID, and we're now the root user *within the container, and only within the container*
- If you try navigating around, you'll notice that it's like being a brand new, fresh install of CentOS – nothing is installed yet
  - The CentOS image is a blank slate we can modify as we please. Changes to the container won't affect our actual server.

Interacting with a Container, Continued

- You can exit from the current container by simply typing *exit*
- After you exit, you can save your changes as a new image using the following command:

```
sudo docker commit -m “<commit message>" -a “<authName>"
<container ID> <reponame>
```

- If you then use the *images* command again, you'll see that your image has been saved as a new image
- Using `sudo docker ps –a`, you can see the docker containers that have already run. Note that they probably all have a status of Exited

★ Using Dockerfiles

- Using the commit interface is one way of building Docker images, but there is a major downside:
  - Docker commit builds off a base image, <u>*and you cannot change that base image without redoing all your work*</u>
- So, what can we do to avoid this problem? Basically, avoid using the commit command at all costs, and use Dockerfiles instead.
- Dockerfiles are easily reproducible, can be updated far easier than an image made with commits, and can be quickly and easily built in Docker

★ Using Dockerfiles

- Using the commit interface is one way of building Docker images, but there is a major downside:
  - Docker commit builds off a base image, *<u>and you cannot change that base image without redoing all your work</u>*
- So, what can we do to avoid this problem? Basically, avoid using the commit command at all costs, and use Dockerfiles instead.
- Dockerfiles are easily reproducible, can be updated far more easily than an image made with commits, and can be quickly and easily built in Docker
- Dockerfiles always use the name 'dockerfile' and have no extension

★ Components of a Dockerfile

- The first declaration in a dockerfile is a *from* declaration: this states what we're using as a base image. In our case, we're using the centos base image:

```
FROM centos
```

- Then, you'll want to use the *run* declaration. Run is used to define the steps we'll use to set up our environment. For example, installing Apache:

```
RUN yum -y install httpd
```

  - This will install Apache and automatically approve the install
  - You will probably have many `RUN` steps within your dockerfile as you build up your environment

★ Copying Files using a Dockerfile

- We can copy files into our Docker image using the *copy* declaration. This is useful for copying project files (such as our Flask application folder), as well as for copying configuration files

```
COPY index.html /var/www/html/index.html
```

- The first file directive is the file on the source system, and the second file directive is the location within the image (in this case, the default Apache document root)
  - Note that Docker uses file locations relative to the dockerfile location – in this case, index.html is stored in the same directory as the dockerfile itself

★ Exposing Ports using Dockerfiles

- Obviously, Docker containers aren't much use to us as web developers if we can't actually 'see' into the container itself
- For this reason, we need to use the *expose* declaration to open up a port in the container that we can pass information through and get responses back. In this example, we'll expose port 80, which is used for HTTP:

```
EXPOSE 80
```

★ Running the Apache Server

- Now we've configured our environment using RUN, copied over any files using COPY, and exposed the necessary port using EXPOSE.
- The final step is to actually start up our server or application within the container, using the *cmd* directive
  - This is actually a little trickier than one might imagine. Many of the commands we normally use to perform this action such as `systemctl start` do not work within Docker
  - We can run Apache by defining it as a foreground process:

```
CMD apachectl –D FOREGROUND
```

★ Building Our Docker Image

- Once you've assembled your dockerfile, you can use it to generate an image. We do this using the docker build command:

```
sudo docker build –t <imageName> <srcdirectory>
```

- The image name can be whatever you want (as long as you avoid conflicting names), and the srcdirectory should be the location of the dockerfile itself
- If everything goes according to plan, you should see Docker running through the steps of building your application, and then it should return "Successfully built <containerID>"

★ Running our Docker Image

- We can run our new Docker image using the docker run command. We'll also add a couple flags to make things easier:
  - The –d flag means "detached". This means the docker instance will run separate from the current shell.
  - The –p flag specifies ports. It allows you to map an input/output port pair. In our case, we'll map the port 80 of the container to port 8080 in the outside world – this way it won't conflict with our existing Apache server
- The final command looks something like this:

```
sudo docker run –p 8080:80 –d <imageName>
```

Docker Instances

- Docker instances are primarily designed to run a single service, not multiple services
  - For example, you'd probably want to separate your database server and web server into two docker instances that communicate remotely, rather than try to run both in one docker instance
- You can run many Docker instances on a single machine, and each will run independently of the others
- Rather than run individual Docker containers, you may also run a Docker Swarm – a distributed cluster of Docker nodes

Interacting with the Docker Registry

- Once you have a completed image, you can use a Docker account to push it to the online Docker registry, making it easy to retrieve. First, use the docker login command to login.
- You'll want to name your image in the following fashion:

```
<dockerUsername>/<repoName>
```

- Then, tag the image using docker tag. The tag may be any single word, and is often used for version information

```
docker tag <dockerUsername>/<repoName>:<tag>
```

- Then, you can just run docker push to push your image to the remote repository:

```
docker push <dockerUsername>/<repoName>:<tag>
```

Reasons for Using Docker

- Docker has enjoyed growing popularity since 2014, and is often a skill that is looked for in hiring practices
- Many services have provided increased integration and support for Docker containers
  - For example, Amazon offers the Elastic Container Service (ECS), which allows you to deploy Docker applications as a scalable cluster
- Excellent for situations where you want to deploy multiple instances of an identical application (for example, load balancing)

Aside: Introduction to Load Balancing

What is Load Balancing?

- Load balancing is a system that uses multiple copies of the same resource (such as a web application) to help handle heavy usage load while avoiding slowdowns or crashes
- For example, many large websites use load balancers to help handle the traffic that is caused by hundreds of thousands of users connecting all at once
  - Even with very high-end servers, there is still an absolute limit to the amount of traffic that can be squeezed through an internet connection

Simple Load Balancing

- We'll focus primarily on the software aspect of load balancing, without getting into the nitty-gritty of the networking hardware that is used for handling serious loads
- Many services offer load balancing as one-click applications: DigitalOcean offers load balancers as a networking option, and Amazon AWS/ECS has multiple ways of producing scaling, balanced applications
- Docker Swarms too can be used as a means of load balancing for a service

Load Balancing Techniques

- Two of the most popular techniques for doing load balancing (without use of a cluster or swarm), is to have multiple server instance ‘slaves’ and a single ‘master’ that simply directs traffic between them
- Traffic may be directed in a round-robin fashion, in which case requests are equally distributed between servers

OR

- The master may keep track of the current load on each server and assign new requests to the server with the least load
- Monitoring introduces an additional degree of complexity, but is a good choice if some services produce additional load. For example, a database query that may bog down a server for some time, while other users simply don’t make that query.

Complications with Load Balancing

- Load balancing introduces several complications to existing web applications. Consider:
  - Your application uses session variables to store data on users between pages. But what if a user's next request ends up being routed to a different server, where the session variables don't exist?
    - Potential solutions: Share sessions using a remote database that all instances access, or simply have the load balancer not re-assign users once they've been assigned the first time
  - Similar issues exist with databases as well. If you're locally hosting your database in each instance, then each instance could have a different version or database state from the others. Create a user on one instance, try to login on the other = error.

Next Week

- In tomorrow's lab, we'll take a look at creating Docker containers and configuring image environments
- Next week, we'll dive a bit more into Docker Swarms, Load Balancing, and general server maintenance practices

That’s All Folks!