2 Sessions

Devops & Deployment

Node Training

Devops & Deployment

- Containerization
- Vanilla Containerized Deployment example
- Managed deployment example
- Production deployment example

Containerization

- Broadly speaking, a container is an execution environment
- Usually the point of the container is to provide a service by exposing some kind of interface
- Types of containers include
  - Bare metal servers
  - Emulated hardware (Virtual Machines)
  - Operating system-level virtualized environments (a.k.a virtual containers)

Containerization

- Bare metal servers
  - most expensive to provision
- Virtual machines
  - much less expensive than bare metal,
  - slowest, most resource intensive option
  - Cross-platform
- Virtualized environments are
  - very cheap
  - very fast
  - OS (kernel) bound

nearForm

Containerization

- Conversations around containerization can often relate specifically to OS-level virtualization (also known as virtualized **containers**)
  - Solaris Zones
  - BSD Jails
  - Linux Containers (LXC's)
- In each case, the OS kernel spawns new user-mode instances, an isolated OS context.
  - Essentially, an advanced form of `chroot`

Containerization

- Linux is by far the most deployed operating system for servers
- Therefore LXC's are the most common form of OS virtualization
- The classic use case of LXC's is shared hosting platforms
- LXC's are (understandably) raw and low-level, working with them requires significant kernel and system level knowledge

Docker

- Docker is a devops tool, originally built on top of LXC - but now built on an LXC-like abstraction using the same primitives as LXC
- Allows for the provision of an exact packaging of a runtime environment
  - Excellent for painless transitions from development to production - everyone's using the same containers
- Cross Operating system compatible
  - The docker CLI on non-Linux systems communicates with a VM running Tiny Core Linux

Docker

- Docker is excellent for any type of project that would benefit from development to production homogeneity
  - e.g. the same environment for both dev and op work
- This tends to range form single process monolithic applications, to distributed architectures such as micro services
- For software services, Docker makes deployment much smoother
- For software products, deliverables can be wrapped in a docker container to ensure a clients environment unconditionally meets expected conditions.

Installing Docker

- Mac OS X
  - https://docs.docker.com/installation/mac/
- Windows
  - https://docs.docker.com/installation/windows/
- Ubuntu
  - https://docs.docker.com/installation/ubuntulinux/
- Or just see
  - https://docs.docker.com/installation/

Docker Processes

- An **image** is an immutable binary file that holds an exact software configuration (such as a vanilla Ubuntu install)
- Docker caches each installed image
- Images are built on top of other images
  - Re-building is very fast
- A **Dockerfile** is the recipe for building an image, generally the Dockerfile would reference a base image to build on top of.

Docker Processes

- When Docker is instructed to run an image, it spawns a **container** (an isolated virtualized environment) which executes the image contents
  - e.g. boots into, say, an ubuntu server which is actually using kernel primitives created by the host machine

The Dockerfile

```
FROM ubuntu
RUN sudo apt-get update
RUN sudo apt-get install -y software-properties-common python
RUN sudo apt-get install -y g++ make git
RUN sudo add-apt-repository ppa:chris-lea/node.js
RUN sudo apt-get update
RUN sudo apt-get install -y nodejs
RUN sudo mv /usr/bin/nodejs /usr/bin/node
```

- The Dockerfile defines the contents of each Image
- Recipe for creating a running system
- The `FROM` instruction specifies an image which can then be built upon
- The resulting image could then be used as a base for another image

nearForm

Key Dockerfile Instructions

- `FROM` - the base image
- `COPY` - copy files from host into container
- `RUN` - execute as part of image set up
- `CMD` - the default command to run on boot
- `USER` - user to run command as
- `EXPOSE` - open a port from inside the container to the outside world
- `ENV` - set environment variables
- `ENTRYPOINT` - the binary that executes commands (default: `/bin/sh`)
- `MAINTAINER` - image maintainer

Docker Exercise

- https://docs.docker.com/userguide/level1
- Complete the Dockerfile quiz to level 2

Key Docker Commands

- **build** - Build an image from a Dockerfile
- **tag** - Tag an image into a repository
- **images** - List images
- **run** - Run a command in a new container
- **ps** - List containers
- **top** - Lookup the running processes of a container
- **attach** - attach (log in) to a running container
- **exec** - execute a command in a running container

Key Docker Commands

- **rm** - Remove one or more containers
- **rmi** - Remove one or more images
- **pull** - Pull an image or a repository from a Docker registry server
- **kill** - Kill a running container
- **restart** - Restart a running container
- **logs** - Fetch the logs of a container
- **commit** - Create a new image from a container's changes
- **diff** - Inspect changes on a container's filesystem

Key Run Flags

```
-d, --detach=false          Detached mode: run the container in the background
                            and print the new container ID

--link=[]                   Add link to another container in the form of
                            name:alias

-p, --publish=[]            Publish a container's port to the host
                            format: ip:hostPort:containerPort |
                            ip::containerPort | hostPort:containerPort |
                            containerPort
                            use 'docker port' to see the actual mapping)

-v, --volume=[]             Bind mount a volume (e.g., from the host: -v /host:/
                             container, from Docker: -v /container)

--volumes-from=[]           Mount volumes from the specified container(s)

-e, --env=[]                Set environment variables

-i, --interactive=false     Keep STDIN open even if not attached
```

nearForm

Container Exercise

- Create two Dockerfiles
- One wraps the `sales-tax-service.js`The other wraps `sales-tax-client.js`Run the containers and get them to talk to each other
- Hint, default Seneca port is `10101`

Example Deployment

```
$ npm install -g yo
$ npm install -g generator-ultimate
$ mkdir mean && cd mean && yo ultimate
```

- We're going to generate a generic MEAN stack using Yeoman
- We'll wrap each of the back end parts in docker containers
- We'll get the docker containers to talk to each other
- We'll add in an nginx container that can load balance multiples web servers

nearForm

Example Deployment

```
$ grunt build
$ docker build -t ultimate-seed .
$ docker run -d --name mongodb
  dockerfile/mongodb
```

- We need to remove Karma from `package.json` (causes failing builds) before the above steps
- We retrieve a fresh MongoDB container from the public docker repo, at the same time as running it

nearForm

Example Deployment

```
$ docker run -d --name redis
  dockerfile/redis
```

```
$ docker logs -f $(docker run -d
  -p 3000:80 --link mongodb:mongodb
  --link redis:redis ultimate-seed)
```

- Ultimate seed also uses redis
- We use the link flag to hook up to our databases
- We capture the container id, and use it tail log output
- On Mac/Windows use `boot2docker ip` to determine the address to make an HTTP request against

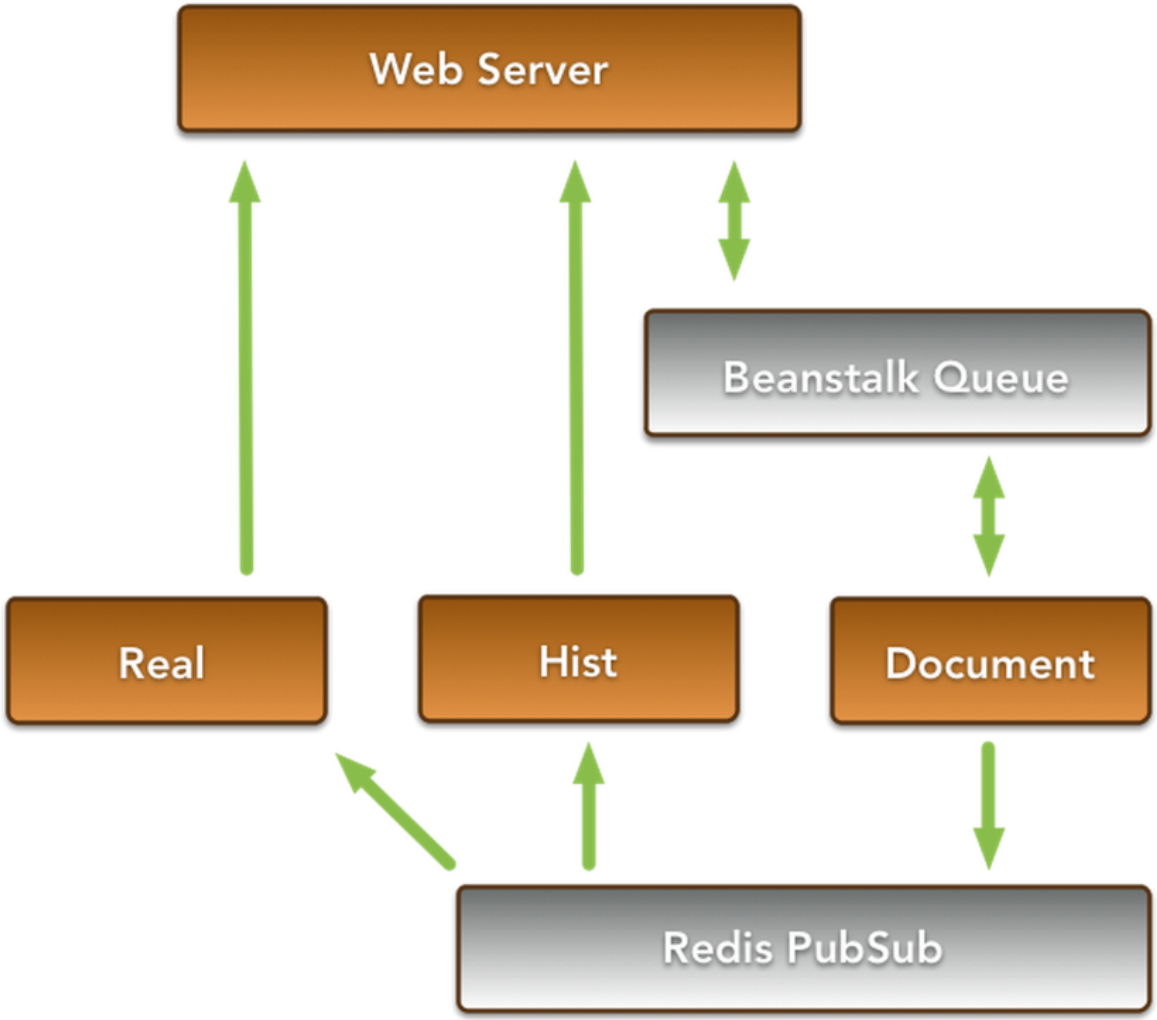
nearForm

Basic Deployment Example

- A basic deployment
- Uses a micro-service architecture
- http://startupdeathclock.com
- http://github.com/rjrodger/startupdeathclock

nearForm

SUDC Architecture



nearForm

SUDC Architecture

- Real, Hist and Document and the web server are Node processes, utilizing the Seneca micro-service framework
- Each Node process is wrapped in it's own docker container
- Redis and Beanstalk are both used to pass messages between micro services
- Both Redis and Beanstalk are also wrapped in separate docker containers

nearForm

Code Structure

- http://github.com/rjrodger/startupdeathclock
- Business logic in `lib` folder
- Service definitions in `srv` folder, which in turn load and use the business logic
- The `contain` directory has multiple sub-directories, each holding a Dockerfile specifying how to build images for each micro-service, the message queue, and Redis
- An additional folder, `contain/base` is the bedrock image of the micro-service images – it's a Dockerfile that installs Node on Ubuntu

Readjustable Deployment

nearForm

Security and Docker

- Versions prior to 1.3.2 are insecure (and unpatchable)
- Prior to the fix, Docker allowed rogue images to write to the host systems during their build process
- This was possible because Docker honored all hard links, which could be crafted to point to the host system during the extraction process
- There was also a bug in 1.3.0-1.3.1 which allowed a malicious image to manipulate security restrictions and potentially break out of the containers context into the host system
- Lesson: don't implicitly trust public images

Security and Docker

- Docker must be run with root privileges
- Docker can run using group privileges, of a group called `docker`, but it must have the same access as root
- This is due the kernel-level nature of Linux containers
  - However future kernel updates may mitigate this
- Lesson: only trusted users should have access to the docker daemon

Security and Docker

- Docker containers do not currently supply root privilege isolation
- Running processes with root privileges inside a container may enable root access to the host system
- Lesson: don't run processes as root in containers
  - If a process must be run as root, use a VM instead, these do provide root safety.
- Docker skirts the issue by providing a capabilities interface, which can be used to enable certain root capabilities in the container, to all non-root users.

Security and Docker

- By design, containers share the same kernel
- Containers therefore can make syscalls to the very same kernel that's currently running the host machine
- If the kernel has a buggy implementation of a syscall, the container can exploit this to break out of the virtualized context
- For instance, around 2008, a bug was introduced into vmsplice (a syscall which joins a PIPE file descriptor to user memory space) whereby proper permissions checks were neglected, opening up a potential overflow exploit
- Lesson: keep Linux servers kernels up to date

Security and Docker

- Set up a DMZ around all deployed containers
- Regardless of the DMZ, consider communication between Docker containers as insecure
- Encrypt transports, e.g. with stunnel

Setting up nscale

```
$ sudo npm -g i nscale
$ git config --global user.name "your un"
$ git config --global user.email "e@ma.il"
```

- **nscale** uses git to track revision history and Github to share/retrieve deployments topologies
- We will need a Github account with SSH keys set up
  - https://help.github.com/articles/generating-ssh-keys/

nearForm

Starting the kernel server

```
$ nscale server start
$ nscale login
```

- The nscale executable is made available by the nscale module at install time
- Almost all nscale commands rely on a remote "kernel" server - we boot the kernel server with nscale server start
- This command will also spin up any other server based nscale services if they're installed - such as the web GUI server
- The nscale login command hooks nscale up with GitHub

Clone a system

```
$ git clone
  git@github.com:nearform/sudc-system.git
```

```
$ nscale system link sudc-system
```

```
$ nscale system list
```

- **nscale** has the concept of a "system", which is a collection of containers representing a topological deployment strategy.
- This is represented in the `system.js` file

nearForm

Compiling and Building

```
$ nscale system compile sudc local
$ nscale container list sudc
$ nscale container buildall sudc
```

- This prepares containers for deployment
- The list command should show the four SUDC containers, web, hist, real and doc along with a root "blank-container"
- Once containers have been built, the system build is added to a revisions list

nearForm

Revisions

```
$ nscale revision list sudc
```

- The revisions list displays changes to the system
- This is the basis for git-like devops management

Deployment Dry Run

```
$ nscale revision preview sudc «rev id»
```

- Use the latest revision id in place of `«rev id»`
  - Assuming there's no clashes between ids the revision id can be the first four or 5 characters

nearForm

SUDC System Topology

```
exports.name = 'sudc-system';
exports.namespace = 'sudc';
exports.id = '62999e58-66a0-4e50-a870-f2673acf6c79';

exports.topology = {
  local: {
    root: ['doc', 'hist', 'real', 'web']
  }
};
```

- The `topology` describes how the system should look,
- In this case we have four containers that are children of a root container
- The definitions for the containers are held in `definitions/services.js`

Creating a System

```
$ nscale system create
$ nscale system list
```

- nscale has the concept of a "system", which is a collection of containers representing a topological deployment strategy.
- Let's name our system "org_sys"

nearForm

Defining Services

definitions/services.js

```
// Place service container definitions here.

exports.root = {
  type: 'container'
};

exports.web = {
  type: 'process',
    specific: {
      repositoryUrl:
        'git@github.com:nearform/nscaledemoweb.git',
      execute: {
        args: '-p 8000:8000 -d',
        exec: '/usr/bin/node index.js'
      }
    }
};
```

nearForm

Defining Mappings

```
// Place project specific type and id mappings here

exports.types = {
};

exports.ids = {
  root: {id: '10'},
  web: {name: 'web'}
};
```

Defining Topology

system.js

```
// Define the system topology here.
// The topology should reference
// containers defined in definitions/services.js

exports.name = 'sherbert';
exports.namespace = 'ilike';
exports.id = '0d2bd8f0-f067-48d6-a4c2-426c4595697d';

exports.topology = {
  local: {
    root: ['web']
  }
};
```

nearForm

Readying for Deploy

```
$ nscale system compile org_sys local
$ nscale container build org_sys web
$ nscale revision list org_sys
```

- System compile uses the JavaScript files we configured, and turns them into JSON
- Revisions are checked with `nscale revision list`
- We prepare for deployment by building the container in this case **nscale** performs a docker build on the docker image.

Readying for Deploy

```
$ nscale system deploy «rev id»
```

- All being well our small system should deploy

nearForm

Production example

```
$ nscale system clone
  http://github.com/rjrodger/nodezoo
```

- Let's take a look at a production example of a microservices deployment using nscale

nearForm

Monitoring

- **node-monitor**: https://github.com/lorenwest/node-monitor
  - Free option source option
  - Involved manual set up process

Monitoring

- **nscale:**
  - Set up a separate nscale server
  - Poll system check
  - Notify and run system fix on fail