Web Servers

This is my current understanding anyway.

A typical Web server registers with the operating system to *listen* on a particular TCP port. When a socket is established, the client will send it's HTTP request. It is the server's responsibility to send the appropriate HTTP response back through the socket (ideally in a secure and efficient manner). That's really all it *has* to do. It often delegates the specifics of *what data* should be contained in the HTTP response to a separate application, such as one consructed using a "web application framework" such as Ruby on Rails.

Apache HTTP Server

Created in 1995, and *took over* the server market (from e.g. Microsoft).

Nginx

Created in 2002, to address the need for a server that can efficiently handle many concurrent requests, for which Apache was unsatisfactory. It does this with an "event driven" model, that involves a single thread serving events emitted by all connections. Seems similar to node.js. As a solution to the concurrent-request problem, nginx was very successful.

Configuration

Reference

- Most of this is a restatement of the information contained in this DigitalOcean tutorial
- Structure notes are from StOve

Structure

- Your config files go in `sites-available/`
- Then to "enable" that site you symlink it into `sites-enabled/`
- By removing the the symlink and issuing `service nginx reload`, one can take down a site
    - With some creativity, one can make it so that if the symlink is not found, a maintenance site is served

Intro to Contexts

- Each `context_type [condition] { block }` defines a "*context*"

- It contains configuration details for *one area of concern*
- The `condition` determines whether to apply the configurations in the `block` to the response to the current request
    - We can think of it as a CSS "selector"
- Context blocks can be nested
- If a value is set inside a `block`, it overrides (i.e. replaces) whatever it was set in a wider scope
- Different setting possibilities are called "*directives*", and each `context_type` only allows the setting of its own set of *directives*
    - Thankfully, if you mess this up, Nginx will exit while initially reading in the configuration file
- The "main" or "global" context is the one outside of all context blocks
    - Correspondingly, use it to modify general behavior of your server
    - E.g. user, group, number of workers, CPU affinity, default error file

Events Context

- Set within the global context (i.e. 1st level of nesting)
- Used to configure how Nginx does its event-based *connection processing*
    - E.g. how many connections each worker can handle, and some specifics about which OS facilities they use to do so

HTTP Context

- Sits at 1st level of nesting in the global context
- Defines defaults for how to handle HTTP[S] connections
    - These may be overriden within more specific contexts
- Configure compression, I/O operations, TCP settings, set document root

Server Context

- Sits within `http` context
- Can be declared multiple times within the same `http` context, without any attached `conditions`
- Each specifies its own "virtual server" that can service a specific subset of connections
- Whether a particular block applies to *this* request depends on the actual "directives" contained *within* the block
    - E.g. the ip-addr:port or the hostname (using the `host` directive)
- Used to configure logging, doc-root, compression, static files, redirects, rewrites, and set (arbitrary) variables

Location context

- Define multiple (like `server` contexts) and "select" for them using the `condition` syntax shown above
- You put them inside `server` contexts, or within other `location`s to add a scope with additional selectivity

Storage

Measuring performance

- Throughput -- how many bytes per second can be transferred
    - This is what most consumers look at, but it is often *not* the system's bottleneck
- Latency -- how long it takes for a data transfer to *begin*
- IOPS -- number of input or output operations per second supported by the device
    - This is generally the bottleneck in the enterprise or data center

References

- throughput vs latency vs iops

Monitoring

Like what?

- Generally time-series data
- Network bandwidth
- temperatures
- CPU load

Database styles

- **RRD** -- *round robin* database based on a *circular buffer*
- **TTL** -- *time to live* database that stores data at a decreasing granularity as it gets older
    - This may involve running further aggregations on it to store it at a lower granularity

Data load models

- **Pull** -- the database polls some intermediate data queue
- **Push** -- the generators issue (e.g.) HTTP POSTs containing JSON data

Tools

- Shai's crayon
- Nagios
- Graphite
- Munin (not that good)
- Ganglia

Ansible

Layman's Terms

- You want to deploy a distributed system
- You tell Ansible how to connect via ssh to each of the member nodes
- Ansible executes the scripts you give it on all of the relevant nodes

Buzzwords

- Ansible is a radically simple IT automation system.
    - It was started in 2012
- Competitors include **Chef** and **Puppet**
    - The major difference being Ansible's '*agentless*' architecture
        - Which means Ansible requires no daemons for background execution
        - This means nodes never poll the control machine
- It manages machines over ssh.
    - That's why it doesn't require *any* software installation on the *managed nodes*
- It handles
    - computer **configuration** & **management**
    - application **deployment**
    - cloud **provisioning**
        - select which machines to use
        - install OS, drivers, middleware, and applications
            - perhaps by using a "boot image"
        - configure system params (e.g. IP address)
    - ad-hoc **task-execution**
    - multinode **orchestration**
        - including trivializing things like zero downtime rolling updates with load balancers.
- Design goals include
    - Minimal dependencies
    - consistency
    - security (esp. of *managed* nodes)
    - reliability (via *idempotent* operations)

    - easy to understand and modify
- Works on all major public and private cloud environments

Basics

- Ansible runs directly from its python source, so upgrading basically amounts to a `git pull`
- **Control machine** -- "could easily be a laptop"
- **Managed nodes** -- require `ssh` to be able to be controlled by the *control machine*
- **Module** -- a standalone (idempotent!) *unit of work* (i.e. script) in Python, Ruby, Bash, etc.
- **Inventory** -- lists IP addrs or hostnames of each accessible node
    - Inventoried nodes may also be assigned to groups
    - If you're using EC2 nodes, look into the specific "EC2 external inventory script"
- **Task** -- a call to an ansible *module*
- **Role** -- calls *tasks*
    - "Roles are great and you should use them every time you write playbooks"
    - They let you combine included files to form clean reusable abstractions
- **Play** -- maps a group of hosts to a set of roles
    - Lists a set of tasks to execute *in order*
- **Playbook** -- a list of *plays* that execute *in order*
    - Expresses configurations, deployment, and orchestration
    - Can launch tasks synchronously *or* asynchronously
    - If a host fails to execute a task, it is not issued any subsequent task for this run of the playbook
        - Shouldn't matter much since playbooks are idempotent, so you can just debug and rerun
    - Run a given playbook YAML file with `ansible-playbook playbook.yml`
- **patterns** -- syntax for telling ansible which hosts the following command applies to
    - `all` or `*` -- target all hosts in the *inventory*
    - `hostname` or `IP.addr` or `groupname` -- target who you think
    - It also gets more complicated if you want it to

Vagrant

- Creates identical development environments repeatably across machines
- after installing it, you run `vagrant init [title]` and it creates a `Vagrantfile`

- In the `Vagrantfile`, you specify how it should retrieve a Linux distro, and how to make a virtual machine in VirtualBox running locally talk to your local (host) machine
- You might now locally run an ansible (or chef, etc.) command to further customize and standardize the environment in that virtual machine
- One of the problems that Vagrant solves is allowing developers to test their changes locally on a system that is configured *exactly* like the production environment
- **Provider** -- your *virtual machine* vendor
    - Defaults to VirtualBox because that's free
    - You may want to install and then use the more stable and performant VMware solution "for any real work"
- **Boxes** -- the package format for Vagrant environments
    - *Many* have been provided by the community
    - They support *versioning*
- **Networking** -- configuration for connecting your guest machine(s) to The Network
    - For example if you have a server running in the virtual machine that you want to connect to from your local/host/laptop's browser, you may want to forward a port like so

      ```
      config.vm.network :forwarded_port, guest: 5601, host: 560
      ```

Debian Packages

- Can be either a *source* or *binary* package
- **Binary** -- distributed using the Debian archive format with suffix `.deb`
    - executables
    - config files
    - man pages
- **Source** -- typically you get one of these to then build the executable locally and then run it
- The package may state other packages on which it depends, replaces, conflicts with, etc.
- [un]installation of debian packages is managed with the `dpkg` software

Elastic Search

Kibana