FabulAWS Documentation

Release 0.3.0a32

Caktus Consulting Group, LLC

Aug 25, 2017

Contents

FabulAWS is a tool for deploying Python web applications to autoscaling-enabled AWS EC2 environments.

CHAPTER 1

Simple example

FabulAWS lets you create EC2 instances using a context manager in Python and easily execute work on that instance. Typical workflow might look like this:

```
from fabulaws.ec2 import MicroLucidInstance

with MicroLucidInstance():
    run('uname -a')
```

If needed, you can extend the instance classes defined in the `fabulaws.ec2` module as needed to further customize the instance before presenting it as a context manager (or using it in your fab file). To do so, simply extend the `setup()` and `cleanup()` methods in one of the existing classes.

CHAPTER 2

Contents

Server Architecture

Prior to creating any new servers or deploying new code using FabulAWS, it's helpful to have an overall understanding of the different components of the server architecture.

FabulAWS creates 5 different types of servers, plus an Amazon Elastic Load Balancer.

Load Balancer

Load balancers are created and managed in the AWS Management Console. The the following ports need to be configured:

- Port 80 forwarded to Port 80 (HTTP)
- Port 443 forwarded to port 443 (HTTPS)

The load balancer health check should be configured as follows (the defaults are fine for the values not listed):

- Ping Protocal: HTTPS
- Ping Port: 443
- Ping Path: `/healthcheck.html`

Web Servers

Web servers are created automatically using FabulAWS. The web servers run Nginx, which proxies a lightweight Gunicorn-powered WSGI server for Django. Also running on the webservers are PgBouncer and Stunnel, which proxy connections to the database master and slave servers, both to speed up connection times and to decrease the load of creating and destroying connections on the actual database servers.

- Sample security groups: myproject-sg, myproject-web-sg

Worker Server

The worker server is very similar in configuration to the web servers, but it runs on a small instance type and does not have Nginx installed. Instead, it is configured to run Celery, the Python package for periodic and background task management. This server is used for tasks like creating response file exports, counting survey start and complete events as they happen, sending out scheduled mailings, and other related tasks. It exists as a separate server to isolate the web servers (which are typically expected to respond very quickly to short requests) from longer-running tasks. Some background tasks may take 5-10 minutes or more to complete.

- Sample security groups: myproject-sg, myproject-worker-sg

Cache and Queue Server

The cache and queue server runs Redis and RabbitMQ. Redis is used both as a cache and an HTTP session storage database. RabbitMQ handles receiving tasks from the web servers and delegating them to the worker server for completion.

- Sample security groups: myproject-sg, myproject-cache-sg, myproject-queue-sg

Database Master

The database master server runs PostgreSQL. It allows encrypted connections from the web and worker servers.

- Sample security groups: myproject-sg, myproject-db-sg

Database Slave

The database slave server also runs PostgreSQL, and is setup with streaming replication from the master database server. This results in very fast (typically less than a few seconds) of lag time between the two machines.

- Sample security groups: myproject-sg, myproject-db-sg

Autoscaling

Each server environment uses EC2 Auto Scaling (AS) to bring up and down new instances based on current demand. When deploying, a new AS Launch Configuration is created for the new revision of the code. The AS Group, which is created and managed largely via the EC2 console, is then updated via the API to point to the new Launch Configuration.

SSL Certificates

SSL certifcates for the production and staging domains can be updated and managed via the Elastic Load Balancers in the AWS console. Internally, the load balancer communicates with the web instances over SSL using the default self-signed certificate that's created on a standard Ubuntu installation (`/etc/ssl/certs/ssl-cert-snakeoil.pem`).

New Project Setup

AWS Configuration

Some configuration within the AWS console is necessary to begin using FabulAWS:

IAM User

First, you'll need to create credentials via IAM that have permissions to create servers in EC2 and manage autoscaling groups and load balancers. Amazon will provide you with a credentials file which will contain `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`, which you will need later in this document.

Security Groups

You'll also need the following security groups. These can be renamed for your project and updated in *fabulaws-config.yml*.

- **myproject-sg**
  - TCP port 22 from 0.0.0.0/0
- **myproject-cache-sg**
  - TCP port 11211 from myproject-web-sg
  - TCP port 11211 from myproject-worker-sg
- **myproject-db-sg**
  - TCP port 5432 from myproject-web-sg
  - TCP port 5432 from myproject-worker-sg
  - TCP port 5432 from myproject-db-sg
- **myproject-queue-sg**
  - TCP port 5672 from myproject-web-sg
  - TCP port 5672 from myproject-worker-sg
- **myproject-session-sg**
  - TCP port 6379 from myproject-web-sg
  - TCP port 6379 from myproject-worker-sg
- **myproject-web-sg**
  - **For EC2-classic:**
    - TCP port 80 from amazon-elb/amazon-elb-sg
    - TCP port 443 from amazon-elb/amazon-elb-sg
  - **For VPC-based AWS accounts:**
    - TCP port 80 from myproject-web-sg
    - TCP port 443 from myproject-web-sg
- **myproject-worker-sg**
  - (used only as a source - requires no additional firewall rules)
- **myproject-incoming-web-sg**
  - TCP port 80 from any address
  - TCP port 443 from any address

Load Balancer

You will need to create a load balancer for your instances, at least one for each environment. Note that multiple load balancers can be used if the site serves different domains (though a single load balancer can be used for a wildcard SSL certificate). Use the following parameters as a guide:

- Choose a name and set it in `fabulaws-config.yml`
- Ports 80 and 443 should be mapped to 80 and 443 on the instances
- If on EC2-Classic (older AWS accounts), you can use 'EC2-Classic' load balancers. Note that this will cause a warning to be shown when you try to 'Assign Security Groups'. That warning can be skipped.
- If on newer, VPC-based AWS accounts:
  - Add security group **myproject-incoming-web-sg** to the load balancer so the load balancer can receive incoming requests.
  - Add security group **myproject-web-sg** to the load balancer so the backend instances will accept forwarded requests from the load balancer.
- Setup an HTTPS health check on port 443 that monitors `/healthcheck.html` at your desired frequency (you'll setup the health check URL in your app below)
- Backend authentication and stickiness should be disabled
- The zones chosen should match those in `fabulaws-config.yml` (typically 2)
- Configure a custom SSL certificate, if desired.

After the load balancer is created, you can set the domain name for the associated environment `fabulaws-config.yml` to your custom domain or the default domain for the load balancer.

Auto Scaling Group

You will also need to create one auto scaling group per envrionment, with the following parameters:

- Choose a name and set it in `fabulaws-config.yml`
- Choose an existing dummy launch config and set it with a "min" and "desired" instances of 0 to start, and a "max" of at least 4 (a higher max is fine).
- Select Advanced, choose your load balancer, and select the ELB health check
- Choose the same availability zones as for your load balancer
- You don't need to configure scaling policies yet, but these will need to be set eventually based on experience
- **You must configure the auto scaling group to tag instances like so:**
  - **Name:** myproject_<environment>_web
  - **deployment:** myproject
  - **environment:** <environment>
  - **role:** web

Local Machine

You'll need to make several changes to your local machine to use FabulAWS:

System Requirements

- Ubuntu Linux 14.04 or later
- Python 2.7
- PostgreSQL 9.3
- virtualenv and virtualenvwrapper are highly recommended

AWS API Credentials

First, you need to define the AWS credentials you created above in your shell environment:

```
export AWS_ACCESS_KEY_ID=...
export AWS_SECRET_ACCESS_KEY=...
```

It's helpful to save these to a file (e.g., `aws.sh`) that you can source (`. aws.sh`) each time they're needed.

Passwords

Local passwords

A number of passwords are required during deployment. To reduce the number of prompts that need to be answered manually, you can use a file called `fabsecrets_<environment>.py` in the top level of your repository.

If you already have a server environment setup, run the following command to get a local copy of fabsecrets_<environment>.py:

```
fab <environment> update_local_fabsecrets
```

Note: If applicable, this will not obtain a copy of the `luks_passphrase` secret which for security's sake is not stored directly on the servers. If you will be creating new servers, this must be obtained securely from another developer.

If this is a brand-new project, you can use the following template for `fabsecrets_<environment>.py`:

```
database_password = ''
broker_password = ''
smtp_password = ''
newrelic_license_key = ''
newrelic_api_key = ''
s3_secret = ''
secret_key = ''
```

All of these are required to be filled in before any servers can be created.

Remote passwords

To update passwords on the server, first retrieve a copy of `fabsecrets_<environment>.py` using the above command (or from another developer) and then run the following command:

```
fab <environment> update_server_passwords
```

Note: It's only necessary to have a copy of `fabsecrets_<environment>.py` locally if you will be deploying new servers or updating the existing passwords on the servers.

Note: This command is really only useful on the web and worker servers. On all other servers, nothing will update the configuration files to use the new secrets.

Project Configuration

You'll need to add several files to your repository, typically at the top level. You can use the following as templates:

fabfile.py

```
import logging

root_logger = logging.getLogger()
root_logger.addHandler(logging.StreamHandler())
root_logger.setLevel(logging.WARNING)

fabulaws_logger = logging.getLogger('fabulaws')
fabulaws_logger.setLevel(logging.INFO)

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)

# XXX import actual commands needed
from fabulaws.library.wsgiautoscale.api import *
```

fabulaws-config.yml

```
instance_settings:
  # http://uec-images.ubuntu.com/releases/trusty/release/
  ami: ami-b2e3c6d8 # us-east-1 14.04.3 LTS 64-bit w/EBS-SSD root store
  key_prefix: 'myproject-'
  admin_groups: [admin, sudo]
  run_upgrade: true
  # Secure directories, volume, and filesystem info
  secure_root: #/secure # no trailing /
  secure_home: #/home/secure
  fs_type: ext4
  fs_encrypt: false
  ubuntu_mirror: us.archive.ubuntu.com
  # create swap of swap_multiplier * available RAM
  swap_multiplier: 1

## REMOTE SETTINGS ##
deploy_user: myproject
webserver_user: myproject-web
database_host: localhost
database_user: dbuser
home: /home/myproject/
```

```
python: /usr/bin/python2.7

## LOCAL / PROJECT SETTINGS ##
disable_known_hosts: true
ssh_keys: deployment/users/
password_names: [database_password, broker_password, smtp_password,
                 newrelic_license_key, newrelic_api_key, s3_secret,
                 secret_key]
project: myproject
wsgi_app: myproject.wsgi:application
requirements_file: requirements/app.txt
requirements_sdists:
settings_managepy: myproject.local_settings
static_html:
  upgrade_message: deployment/templates/html/503.html
  healthcheck_override: deployment/templates/html/healthcheck.html
localsettings_template: deployment/templates/local_settings.py
logstash_config: deployment/templates/logstash.conf

# Set gelf_log_host to the host of your Graylog2 server (or other GELF log
# receiver)
# gelf_log_host: hostname

# Set syslog_server to a "hostname:port" (quote marks required due
# to the ":" in there) and server logs will be forwarded there using
# syslog protocol.  "hostname:port" could be e.g. papertrail or a
# similar service.
# (You might want to set this in fabsecrets instead of here.)
# syslog_server: "hostname:port"

# You can alternatively supply a multi-line config for rsyslog as follows
# (e.g., in the event you need to enable TLS). For more information, see:
# http://www.rsyslog.com/doc/v8-stable/tutorials/tls_cert_client.html#sample-syslog-
↪conf
# syslog_server: |
#   # make gtls driver the default
#   $DefaultNetstreamDriver gtls
#
#   # certificate files
#   $DefaultNetstreamDriverCAFile /rsyslog/protected/ca.pem
#   $DefaultNetstreamDriverCertFile /rsyslog/protected/machine-cert.pem
#   $DefaultNetstreamDriverKeyFile /rsyslog/protected/machine-key.pem
#
#   $ActionSendStreamDriverAuthMode x509/name
#   $ActionSendStreamDriverPermittedPeer central.example.net
#   $ActionSendStreamDriverMode 1 # run driver in TLS-only mode
#   *.* @@central.example.net:10514 # forward everything to remote server

# Set awslogs_access_key_id to the AWS_ACCESS_KEY_ID of the user with
# permissions to create log groups, log streams, and log events. For help
# setting up this role, see:
# http://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/QuickStartEC2Instance.html
# NOTE: You will also need to set awslogs_secret_access_key in your
# fabsecrets_<environment>.py file
# awslogs_access_key_id: AK....

# Set extra_log_files to a list of log files you want to monitor, in addition
# to the default logs monitored by FabulAWS itself:
```

```
# extra_log_files:
#   /path/to/file:
#     tag: mytag
#     date_format: '%Y-%m-%d %H:%M:%S'

backup_key_fingerprint:
vcs_cmd: git # or hg
latest_changeset_cmd: git rev-parse HEAD # hg id -i # or git rev-parse HEAD
repo: git@github.com:username/myproject.git
# Mapping of Fabric deployments and environments to the Mercurial branch names
# that should be deployed.
branches:
  myproject:
    production: master
    staging: master
    testing: master

## SERVER SETTINGS ##

# Local server port for pgbouncer
pgbouncer_port: 5432

# Version of Less to install
less_version: 2.5.3

# Local server ports used by Gunicorn (the Django apps server)
server_ports:
  staging: 8000
  production: 8001
  testing: 8002

# Whether we're hosting static files on our webservers ('local')
# or somewhere else ('remote')
static_hosting: remote

# Mapping of celery worker names to options
# The worker name (key) can be any text of your choosing. The value should
# be any additional options you'd like to pass to celeryd, such as specifying
# the concurrency and queue name(s)
celery_workers:
  main: -c 10 -Q celeryd

# Start this many Gunicorn workers for each CPU core
gunicorn_worker_multiplier: 8

# Mapping of environment names to domain names. Used to update the
# primary site in the database after a refresh and to set ALLOWED_HOSTS
# Note that the first domain in the list must not be a wildcard as it
# is used to update a Site object in the database.
# Wildcard format used per ALLOWED_HOSTS setting
site_domains_map:
  production:
  - dualstack.myproject-production-1-12345.us-east-1.elb.amazonaws.com
  staging:
  - dualstack.myproject-staging-1-12345.us-east-1.elb.amazonaws.com
  testing:
  - dualstack.myproject-testing-1-12345.us-east-1.elb.amazonaws.com
```

```
## ENVIRONMENT / ROLE SETTINGS ##

default_deployment: myproject
deployments:
- myproject
environments:
- staging
- production
- testing
production_environments:
- production
valid_roles:
- cache
- db-master
- db-slave
- web
- worker

## AWS SETTINGS ##

region: us-east-1
avail_zones:
- e
- c

# Mapping of role to security group(s):
security_groups:
  db-master: [myproject-sg, myproject-db-sg]
  db-slave: [myproject-sg, myproject-db-sg]
  cache: [myproject-sg, myproject-session-sg, myproject-cache-sg, myproject-queue-sg]
  worker: [myproject-sg, myproject-worker-sg]
  web: [myproject-sg, myproject-web-sg]

# Mapping of environment and role to EC2 instance types (sizes)
instance_types:
  production:
    cache: c3.large
    db-master: m3.xlarge
    db-slave: m3.xlarge
    web: c3.large
    worker: m3.large
  staging:
    cache: t1.micro
    db-master: m1.small
    db-slave: m1.small
    web: m1.small
    worker: m3.large
  testing:
    cache: t1.micro
    db-master: t1.micro
    db-slave: t1.micro
    web: m1.small
    worker: m1.small

# Mapping of Fabric environment names to AWS load balancer names.  Load
# balancers can be configured in the AWS Management Console.
load_balancers:
  myproject:
```

```
    production:
    - myproject-production-lb
    staging:
    - myproject-staging-lb
    testing:
    - myproject-testing-lb

# Mapping of Fabric environment names to AWS auto scaling group names. Auto
# scaling groups can be configured in the AWS Management Console.
auto_scaling_groups:
  myproject:
    production: myproject-production-ag
    staging: myproject-staging-ag
    testing: myproject-testing-ag

# Mapping of Fabric environment and role to Elastic Block Device sizes (in GB)
volume_sizes:
  production:
    cache: 10
    db-master: 100
    db-slave: 100
    web: 10
    worker: 50
  staging:
    cache: 10
    db-master: 100
    db-slave: 100
    web: 10
    worker: 50
  testing:
    cache: 10
    db-master: 100
    db-slave: 100
    web: 10
    worker: 50

# Mapping of Fabric environment and role to Elastic Block Device volume types
# Use SSD-backed storage (gp2) for all servers. Change to 'standard' for slower
# magnetic storage.
volume_types:
  cache: gp2
  db-master: gp2
  db-slave: gp2
  web: gp2
  worker: gp2

app_server_packages:
  - python2.7-dev
  - libpq-dev
  - libmemcached-dev
  - supervisor
  - mercurial
  - git
  - build-essential
  - stunnel4
  - pgbouncer

db_settings:
```

```
  # for help adjusting these settings, see:
  # http://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server
  # http://wiki.postgresql.org/wiki/Number_Of_Database_Connections
  # http://thebuild.com/presentations/not-my-job-djangocon-us.pdf
  postgresql_settings:
    # Settings to apply to Postgres servers
    # You can put anything here from postgresql.conf

    # connections
    max_connections: '80' # _active_ connections are limited by pgbouncer

    # replication settings
    wal_level: 'hot_standby'
    hot_standby: 'on'
    hot_standby_feedback: 'on'
    max_wal_senders: '3'
    wal_keep_segments: '3000' # during client deletion 50 or more may be generated␣
↪per minute; this allows an hour

    # resources - let pgtune set these based on actual machine resources
    # shared_buffers: '8GB' # 25% of available RAM, up to 8GB
    # work_mem: '750MB' # (2*RAM)/max_connections
    # maintenance_work_mem': '1GB' # RAM/16 up to 1GB; high values aren't that helpful
    # effective_cache_size': '48GB' # between 50-75%, should equal free + cached␣
↪values in `top`

    # checkpoint settings
    wal_buffers: '16MB'
    checkpoint_completion_target: '0.9'
    checkpoint_timeout: '10min'
    checkpoint_segments: '256' # if checkpoints are happening more often than the␣
↪timeout, increase this up to 256

    # logging
    log_min_duration_statement: '500'
    log_checkpoints: 'on'
    log_lock_waits: 'on'
    log_temp_files: '0'

    # write optimizations
    commit_delay: '4000' # delay each commit this many microseconds in case we can do␣
↪a group commit
    commit_siblings: '5' # only delay if at least N transactions are in process

    # index usage optimizations
    random_page_cost: '2' # our DB servers have a lot of RAM and may tend to prefer␣
↪Seq Scans if this is too high

  # More Postgres-related settings.
  # How to install Postgres:
  postgresql_packages:
    - postgresql
    - libpq-dev
  # Whether and how to apply pgtune
  postgresql_tune: true
  postgresql_tune_type: Web
  # Kernel sysctl settings to change
  postgresql_shmmax: 107374182400  # 100 GB
```

```
  postgresql_shmall: 26214400  # 100 GB / PAGE_SIZE (4096)
  # Networks to allow connections from
  postgresql_networks:
    - '10.0.0.0/8'
    - '172.16.0.0/12'
  # Whether to disable the Linux out-of-memory killer
  postgresql_disable_oom: true
```

local_settings.py

This file should be placed at the location specified in `fabulaws-config.yml`, typically `deployment/templates/local_settings.py`.

```
from myproject.settings import *

DEBUG = False

# logging settings
#LOGGING['filters']['static_fields']['fields']['deployment'] = '{{ deployment_tag }}'
#LOGGING['filters']['static_fields']['fields']['environment'] = '{{ environment }}'
#LOGGING['filters']['static_fields']['fields']['role'] = '{{ current_role }}'
AWS_STORAGE_BUCKET_NAME = '{{ staticfiles_s3_bucket }}'
AWS_ACCESS_KEY_ID = 'YOUR-KEY-HERE'
AWS_SECRET_ACCESS_KEY = "{{ s3_secret }}"

SECRET_KEY = "{{ secret_key }}"

# Tell django-storages that when coming up with the URL for an item in S3 storage,␣
↪keep
# it simple - just use this domain plus the path. (If this isn't set, things get␣
↪complicated).
# This controls how the `static` template tag from `staticfiles` gets expanded, if you
↪'re using it.
# We also use it in the next setting.
AWS_S3_CUSTOM_DOMAIN = '%s.s3.amazonaws.com' % AWS_STORAGE_BUCKET_NAME

# This is used by the `static` template tag from `static`, if you're using that. Or␣
↪if anything else
# refers directly to STATIC_URL. So it's safest to always set it.
STATIC_URL = "https://%s/" % AWS_S3_CUSTOM_DOMAIN

# Tell the staticfiles app to use S3Boto storage when writing the collected static␣
↪files (when
# you run `collectstatic`).
STATICFILES_STORAGE = 'storages.backends.s3boto.S3BotoStorage'

# Auto-create the bucket if it doesn't exist
AWS_AUTO_CREATE_BUCKET = True

AWS_HEADERS = {  # see http://developer.yahoo.com/performance/rules.html#expires
    'Expires': 'Thu, 31 Dec 2099 20:00:00 GMT',
    'Cache-Control': 'max-age=94608000',
}

# Having AWS_PRELOAD_META turned on breaks django-storages/s3 -
# saving a new file doesn't update the metadata and exists() returns False
```

```
#AWS_PRELOAD_METADATA = True

# database settings
DATABASES = {
{% for server in all_databases %}
    '{{ server.database_key }}': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': '{{ server.database_local_name }}',
        'USER': '{{ database_user }}',
        'PASSWORD': '{{ database_password }}',
        'HOST': 'localhost',
        'PORT': '{{ pgbouncer_port }}',
    },{% endfor %}
}

# django-balancer settings
DATABASE_POOL = {
{% for server in slave_databases %}
    '{{ server.database_key }}': 1,{% endfor %}
}
MASTER_DATABASE = '{{ master_database.database_key }}'

# media roots
MEDIA_ROOT = "{{ media_root }}"
STATIC_ROOT = "{{ static_root }}"

# email settings
EMAIL_HOST_PASSWORD = '{{ smtp_password }}'
EMAIL_SUBJECT_PREFIX = '[{{ deployment_tag }} {{ environment }}] '

# Redis DB map:
# 0 = cache
# 1 = unused (formerly celery task queue)
# 2 = celery results
# 3 = session store
# 4-16 = (free)

# Cache settings
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '{{ cache_server.internal_ip }}:11211',
        'VERSION': '{{ current_changeset }}',
    },
    'session': {
        'BACKEND': 'redis_cache.RedisCache',
        'LOCATION': '{{ cache_server.internal_ip }}:6379',
        'OPTIONS': {
            'DB': 3,
        },
    },
}

# Task queue settings

# see https://github.com/ask/celery/issues/436
BROKER_URL = "amqp://{{ deploy_user }}:{{ broker_password }}@{{ cache_server.internal_
↪ip }}:5672/{{ vhost }}"
```

```
BROKER_CONNECTION_TIMEOUT = 4
BROKER_POOL_LIMIT = 10
CELERY_RESULT_BACKEND = "redis://{{ cache_server.internal_ip }}:6379/2"

# Session settings
SESSION_ENGINE = 'django.contrib.sessions.backends.cache'
SESSION_CACHE_ALIAS = 'session'

# django-compressor settings
COMPRESS_URL = STATIC_URL
# Use MEDIA_ROOT rather than STATIC_ROOT because it already exists and is
# writable on the server.
COMPRESS_ROOT = MEDIA_ROOT
COMPRESS_STORAGE = STATICFILES_STORAGE
COMPRESS_OFFLINE = True
COMPRESS_OFFLINE_MANIFEST = 'manifest-{{ current_changeset }}.json'
COMPRESS_ENABLED = True

ALLOWED_HOSTS = [{% for host in allowed_hosts %}'{{ host }}', {% endfor %}]
```

SSH keys

Before attempting to deploy for the first time, you should add your SSH public key to a file named `deployment/users/<yourusername>` in the repository. This path can also be configured in `fabulaws-config.yml`. Multiple SSH keys are permitted per file, and additional files can be added for each username (developer).

Django Settings

FabulAWS uses django_compressor and django-storages to store media on S3. The following settings changes are required in your base `settings.py`:

1. `compressor`, `storages`, and `djcelery` should be added to your `INSTALLED_APPS`.
2. Add the following to the end of your `settings.py`, modifying as needed:

```
# Celery settings
import djcelery
from celery.schedules import crontab
djcelery.setup_loader()

CELERY_SEND_TASK_ERROR_EMAILS = True

# List of finder classes that know how to find static files in
# various locations.
STATICFILES_FINDERS = (
    'django.contrib.staticfiles.finders.FileSystemFinder',
    'django.contrib.staticfiles.finders.AppDirectoriesFinder',
    'compressor.finders.CompressorFinder',
)

STATIC_ROOT = os.path.join(BASE_DIR, 'static')

COMPRESS_ENABLED = False # enable in local_settings.py if needed
COMPRESS_CSS_HASHING_METHOD = 'hash'
COMPRESS_PRECOMPILERS = (
```

```
    ('text/less', 'lessc {infile} {outfile}'),
)
```

wsgi.py

You'll need to change the default `DJANGO_SETTINGS_MODULE` in your project's `wsgi.py` to `myproject.local_settings`.

Static HTML

You need to create two static HTML files, one for displaying an upgrade message while you're deploying to your site, and one to serve as a "dummy" health check to keep instances in your load balancer healthy while deploying.

The paths to these files can be configured in the `static_html` dictionary in your `fabulaws-config.yml`:

```
static_html:
  upgrade_message: deployment/templates/html/503.html
  healthcheck_override: deployment/templates/html/healthcheck.html
```

The `503.html` file can contain anything you'd like. We recommend something distictive so that you can tell if your health check is being served by Django or the "dummy" health check html file, e.g.: `OK (nginx override)`

Similarly, the `healthcheck.html` can contain anything you'd like, either something as simple as `Upgrade in progress. Please check back later.` or a complete HTML file complete with stylesheets and images to display a "pretty" upgrade-in-progress message.

Basic Auth

If you want to add HTTP Basic Auth to a site, add a section to `fabulaws-config.yml` like this:

```
# Any sites that need basic auth
# This is NOT intended to provide very high security.
use_basic_auth:
  testing: True
  anotherenv: True
```

Add `basic_auth_username` and `basic_auth_password` to `password_names`:

```
password_names: [a, b, c, ..., basic_auth_username, basic_auth_password]
```

And add the desired username and password to each environment secrets file:

```
basic_auth_username: user1
basic_auth_password: password1
```

You'll need to add these entries to all secrets files; just set them to an empty string for environments where you are not using basic auth.

Then in the `testing` and `anotherenv` environments, fabulaws will apply basic auth to the sites. For testing, user `user1` will be able to use password `password1`, and so forth.

Note: Fabulaws will also turn off Basic Auth for the health check URL so that the load balancer can access it. It assumes that the health check URL is `/healthcheck.html` and that Django will be serving the health check URL

(rather than being served as a static file directly by Nginx, for example). If either of those assumptions are not correct, you will need to tweak it by copying and modifying the template for nginx.conf.

Health Check

You'll need to configure a health check within Django as well. Following is a sample you can use.

Add to `views.py`:

```
import logging

from django.db import connections
from django.http import HttpResponse, HttpResponseServerError


def health_check(request):
    """
    Health check for the load balancer.
    """
    logger = logging.getLogger('fabutest.views.health_check')
    db_errors = []
    for conn_name in connections:
        conn = connections[conn_name]
        try:
            cursor = conn.cursor()
            cursor.execute('SELECT 1')
            row = cursor.fetchone()
            assert row[0] == 1
        except Exception, e:
            # note that there doesn't seem to be a way to pass a timeout to
            # psycopg2 through Django, so this will likely not raise a timeout
            # exception
            logger.warning('Caught error checking database connection "{0}"'
                           ''.format(conn_name), exc_info=True)
            db_errors.append(e)
    if not db_errors:
        return HttpResponse('OK')
    else:
        return HttpResponseServerError('Configuration Error')
```

Add lines similar to those highlighted below to your `urls.py`:

```
from django.conf.urls import include, url
from django.contrib import admin

from fabutest import views as fabutest_views

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^healthcheck.html$', fabutest_views.health_check),
]
```

Python Requirements

The following are the minimum Python requirements for deploying a web application using FabulAWS (update version numbers as needed):

```
Django==1.8.8
psycopg2==2.6.1
pytz==2015.7
django-celery==3.1.17
celery==3.1.19
gunicorn==19.4.5
django-balancer==0.4
boto==2.39.0
django-storages==1.1.8
django-compressor==2.0
python-memcached==1.57
redis==2.10.5
django-redis-cache==1.6.5
django-cache-machine==0.9.1
newrelic==2.60.0.46
```

In addition, the following requirements are needed for deployment:

```
fabric==1.10.2
boto==2.39.0
pyyaml==3.11
argyle==0.2.1
```

First Deployment

Once you have your EC2 environment and project configured, it's time to create your initial server environment.

To create a new instance of the testing environment, you can use the `create_environment` command to Fabric, like so:

```
fab create_environment:myproject,testing
```

In addition to the console, be sure to inspect the log files generated (`*.out` in the current directory) to troubleshoot any problems that may arise.

For more information, please refer to the *Deployment* documentation.

Deployment

FabulAWS uses Fabric for deployment, with which some familiarity is strongly recommended. This page assumes that you've completed the necessary setup described in the *Server Architecture*. You will also need a local development environment setup as described in *New Project Setup*.

Important: When deploying to an environment, your local copy of the code should be up to date and must also have a checkout of the correct branch. Environments can be mapped to branches in the `fabulaws-config.yml` file.

Testing environment

The **testing environment** is usually a temporary environment that can be used for load testing, deployment testing, or other activities that would otherwise negatively impact the primary environment for testing new features (see *Staging environment* below).

To create a new instance of the testing environment, you can use the `create_environment` command to Fabric, like so:

```
fab create_environment:myproject,testing
```

Prior to running this command, be certain that all prior instances of testing servers have been terminated via the AWS console. This command will create all the required servers in parallel. To avoid difficulty with determining which server failed to be created when a problem is encountered, the logs for server creation are saved to separate files. *Always check these files to ensure that the servers were created successfully.*

Staging environment

The **staging environment** is typically used for testing and quality assurance of new features. It also serves as a testing ground for doing the deployment itself. New features (even small bug fixes) should be deployed to and tested on the staging environment prior to being deployed to the production environment.

The staging environment is usually a copy of the production environment running on smaller (cheaper) virtual machines at EC2. It also typically contains a recent snapshot of the production database, so any issues specific to the production environment can be tested on staging without affecting usability of the production site.

Production environment

The **production environment** is typically hosts the live servers in use by the the application's end-users.

Deployment methods

Since FabulAWS uses Amazon Autoscaling, special care must be taken to update the autoscaling image at the same time as new code is deployed.

Autoscaling: Updating the image

Because AMI creation can be a time-intensive part of the process, it can be done separately ahead of time to prepare for a deployment.

To create an autoscaling AMI and launch configuration based on the current version of the code (from the appropriate branch - see above), run the following command:

```
fab create_launch_config_for_deployment:myproject,<environment>
```

This command will print out the name of the created launch configuration, which can be passed into the associated autoscaling deployment methods below. If needed, the launch configuration names and associated images can also be found via the AWS console.

Autoscaling: Full deployment

A "full" deployment should be used any time there are backwards-incompatible updates to the application, i.e., when having two versions of the code running simultaneously on different servers might have damaging results or raise errors for users of the site. Note that this type of deployment requires downtime, which may need to be scheduled ahead of time depending on which environment is impacted.

With autoscaling, a full deployment works as follows:

1. First, the autoscaling group's ability to add new instances to the load balancer is suspended, a new launch configuration for the new version of the code is installed, and the desired number of instances for the group is doubled. This has the effect of spinning up all the new required instances without adding them to the load balancer.
2. Once those instances have been created, the "upgrade in progress" message is displayed on *all* the servers, `deploy_worker` is run to update the database schema and any static media, and the autoscaling group's ability to add instances to the load balancer is resumed. The process then waits for all instances to be healthy in the load balancer.
3. Finally, the old instances in the group are terminated, and the "upgrade in progress" message is removed from the new servers.

The syntax for completing a full deployment is as follows:

```
fab deploy_full:myproject,<environment>[,<launch config name>]
```

The launch configuration name is optional, and one will be created automatically if not specified.

Note: This command does not update secrets from your local file to the servers. If you want to do that, explicitly run `fab <environment> update_server_passwords` before running this command.

Autoscaling: Serial deployment

A "serial" deployment can be used any time the changes being deployed are minimal enough that having both versions of the code running simultaneously will not cause problems. This is usually the case any time there are minor, code-only (non-schema) updates. Each server points to a separate copy of the static media specific to the version of the code that it's running, so backwards incompatible CSS and JavaScript changes can safely be deployed serially.

Serial deployments with autoscaling work by gradually marking instances in the autoscaling group as unhealthy, and then waiting for the group to create a new, healthy instance before proceeding. A serial deployment can be started as follows:

```
fab deploy_serial:cmyproject,<environment>[,<launch config name>]
```

Again, the launch config is optional and one will be created automatically if not specified.

Note: This command does not update secrets from your local file to the servers. If you want to do that, explicitly run `fab <environment> update_server_passwords` before running this command.

Note: You may see errors that look like this while running a serial deployment:

```
400 Bad Request
<ErrorResponse xmlns="http://elasticloadbalancing.amazonaws.com/doc/2012-06-01/">
```

```
  <Error>
    <Type>Sender</Type>
    <Code>InvalidInstance</Code>
    <Message>Could not find EC2 instance i-1bb70c35.</Message>
  </Error>
  <RequestId>9b3dc6a5-850e-11e3-9e35-b9e8294315ba</RequestId>
</ErrorResponse>
```

These errors are expected and simply mean that the elastic load balancer is not yet aware of the newly created instance.

Suspending and restarting autoscaling processes

If for any reason autoscaling needs to be suspended, this can be accomplished through Fabric. To suspend all autoscaling processes, simply run:

```
fab suspend_autoscaling_processes:myproject,<environment>
```

To resume autoscaling once any issues have been resolved, run:

```
fab resume_autoscaling_processes:myproject,<environment>
```

A note about usernames

If you get a prompt that looks something like this when you attempt to deploy, it's quite possible that you're giving the remote server the wrong username (or you don't have access to the servers to begin with):

```
[ec2-23-22-145-188.compute-1.amazonaws.com] Passphrase for private key:
```

When deploying to any environment, if your local username is different from the username you use to login to the remote server, you need to give Fabric a username on the command line, like so:

```
fab -u <remoteusername> <environment> <commands>
```

Maintenance Tasks

Describing an environment

While performing maintenance on an environment, it's sometimes helpful to know exactly what servers are in that environment and what their load balancer status is, if any. To get a list of all servers in a given environment and print some basic meta data about those servers, you can use the `describe` command to Fabric, like so:

```
fab describe:myproject,<environment>
```

Adding new sysadmin users

If you don't have access to the servers yet, add your SSH public key in the deployment/users/ directory. To avoid having to pass a -u argument to fabric on every deploy, make the name of the file identical to your local username. Then ask someone who has access to run this command:

```
fab staging update_sysadmin_users
```

Updating New Relic keys

To update the New Relic API and License keys, first find the new keys from the new account. The License Key can be found from the main account page, and the API key can be found via these instructions: https://docs.newrelic.com/docs/apis/api-key

Next, make sure your local fabsecrets_<environment>.py file is up to date:

```
fab production update_local_fabsecrets
```

Next, update the `newrelic_license_key` and `newrelic_api_key` values inside the `fabsecrets_<environment>.py` file with the new values. Then, update the keys on the servers:

```
fab staging update_server_passwords
fab production update_server_passwords
```

Finally, update the configuration files containing the New Relic keys and restart the Celery and Gunicorn processes:

```
fab update_newrelic_keys:myproject,staging
fab update_newrelic_keys:myproject,production
```

Note this short method of updating the configuration files involves a brief moment of downtime (10-20 seconds). If no downtime is desired, you can achieve the same result by repeating the following commands for each environment, as needed (but it will take much longer, i.e., 30-60 minutes):

```
fab production upload_newrelic_sysmon_conf
fab production upload_newrelic_conf
fab deploy_serial:myproject,production
```

Copying the database from production to staging or testing

To copy the production database on the staging server, run the following command:

```
fab staging reload_production_db
```

This will drop the current staging DB, create a new database, load it with a copy of the current production data, and then run any migrations not yet run on that database. The same command will work on the testing environment by replacing "staging" with "testing". Internally, autoscaling is suspended and an upgrade message is displayed on the servers while this command is in progress.

Fixing an issue with broken site icons

If the button icons on the site appear as text rather than as images, there is probably an issue with the CORS configuration for the underlying S3 bucket that serves the font used to show these icons. To correct this, follow these steps:

First, navigate to the S3 bucket in the AWS Console, and click the Properties tab

Next, expand the Permissions section and then click Add CORS Configuration. The text in the popup should look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
    <CORSRule>
        <AllowedOrigin>*</AllowedOrigin>
        <AllowedMethod>GET</AllowedMethod>
        <MaxAgeSeconds>3000</MaxAgeSeconds>
        <AllowedHeader>Authorization</AllowedHeader>
    </CORSRule>
</CORSConfiguration>
```

Finally, click the Save button to add the configuration. **This step is important; while it may appear that the configuration is already correct, it needs to be saved before it will be added by S3.**

Stopping EC2 machines while not in use

Some types of instances, included db-master, db-slave, and worker servers, can be stopped via the AWS console, later restarted, and then reconfigured by running the following commands (in order):

```
fab <environment> mount_encrypted:roles=db-master
fab <environment> mount_encrypted:roles=db-slave
fab <environment> mount_encrypted:roles=worker
```

The cache server, due to an intricacy with how RabbitMQ stores its data and configuration files, must be completely terminated and recreated (it does not support changing the host's IP address). For more information, see: http://serverfault.com/questions/337982/how-do-i-restart-rabbitmq-after-switching-machines

Web servers are managed via Amazon Auto Scaling. To terminate all web servers, simply navigate to the AWS Auto Scaling Group and set the Minimum, Desired, and Maximum number of instances to zero. Failure to complete this step may result in the Auto Scaling Group perpetually attempting to bring up new web servers and failing because no database servers exist.

Resizing servers or recreating an environment

An entire environment can be recreated, optionally with different server sizes, with a single command. Note that this command takes a long time to run (30-60 minutes or even several hours, depending on the size of the database). For this reason, it is beneficial to clean out the database (see above) before downsizing the servers because copying the database from server to server takes a significant portion of this time. That said, the environment will not be down or inaccessible for this entire time; rather, the script does everything in an order that minimizes the downtime required. For a typical set of smaller servers and an empty database, the downtime will usually be less than 2 minutes.

If you'd like to resize an environment, first edit the `instance_types` dictionary in `fabulaws-config.yml` to the sizes you'd like for the servers. Here are the minimum sizes for each server type:

- cache: `m1.small`
- db-master: `m1.small`
- db-slave: `m1.small`
- web: `m1.small`
- worker: `m1.medium`

Once the sizes have (optionally) been adjusted, you can recreate the environment like so:

```
fab recreate_servers:myproject,production
```

Updating Dependencies

To circumvent the inevitable issues with PyPI during deployment, sdists for all dependencies needed in the staging and production environments must be added to the `requirements/sdists/` directory. This means that, whenever you change in `requirements/apps.txt`, you should make a corresponding change to the `requirements/sdists/` directory.

Adding or updating a single package

To download a single sdist for a new or updated package, run the following command, where `package-name==0.0.0` is a copy of the line that you added to `requirements/apps.txt`:

```
pip install package-name==0.0.0 -d requirements/sdists/
```

After downloading the new package, remove the outdated version from version control, and add the new one along with the change to apps.txt.

Repopulating the entire sdists/ directory

You can also repopulate the entire sdists directory as follows:

```
cd requirements/
mkdir sdists_new/
pip install -r apps.txt -d sdists_new/
rm -rf sdists/
mv sdists_new/ sdists/
```

Upgrading system packages

Since the site uses Amazon Auto Scaling, to ensure the servers have the latest versions of Ubuntu packages we first need to update the web server image. This can be done by running a new deployment, like so:

```
fab deploy_serial:myproject,<environment>
```

Upgrading Ubuntu packages on the persistent (non-web) servers can be done with the `upgrade_packages` Fabric command. Before upgrading, it's best to take the site offline and put it in upgrade mode to avoid any unexpected error pages while services are restarted:

```
fab <environment> begin_upgrade
```

Once the site is in upgrade mode, you can update packages on the servers as follows:

```
fab <environment> upgrade_packages
```

This command will connect to the servers one by one, run `apt-get update`, install any new packages needed by the web servers, and then run `apt-get upgrade`. You will be prompted to accept any upgrades that need to take place, so you will have the opportunity to cancel the upgrade if needed for any reason.

After verifying that the packages have installed successfully, you can bring the site back online like so:

```
fab <environment> end_upgrade
```

Note that upgrading may take some time, depending on the number of servers and size of the upgrades, so it's best to schedule this during an off-hours maintenance window.

Troubleshooting server issues

Managing SSH host keys

Amazon will regularly reuse IP addresses for servers, which can cause conflicts with your local ssh host keys (`~/.ssh/known_hosts` on most systems). If you see a message like this while creating a server, you'll know you're affected by this:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
```

If you see this message, you will need to terminate the server being created, delete the host key (for both the hostname and the IP address) from your `known_hosts` file, and re-create the server.

Resetting a troubled environment

If all the servers and services in an environment appear to be running properly but the web frontend or worker still isn't functioning, one of the quickest ways to "reset" an environment is to run a deployment. Before attempting more extreme measures, you can run a deployment to get all the config files in sync and restart many of the services that make up the application, like so:

```
fab <environment> begin_upgrade deploy_worker deploy_web end_upgrade
```

Fixing Celery when it won't stop

Celery occassionally gets into a state when it won't stop and may be pegging the CPU. If this happens (e.g., if the `deploy_worker` command hangs indefinitely while stopping the Celery workers), you may need to SSH to the worker server manually and run:

```
sudo killall -9 celery
```

If this doesn't work, you can revert to manually finding the PIDs of the stuck Celery processes in `ps auxww` or `top` and killing them with:

```
sudo kill -9 <PID>
```

After doing this, be sure to run `deploy_worker` (or if it was already running, let it complete) so as to restore Celery to a running state again.

Master database goes down

If the master database goes down, manually make sure it's permanently lost before converting a slave into the master. At this point you probably also want to enable the pretty upgrade message on all the web servers:

```
fab <environment> begin_upgrade
```

Any slave can be "promoted" to the master role via fabric, as follows:

```
fab <environment> promote_slave
```

This will tag the old master as "decommissioned" in AWS, tag the slave as the new master, and then run the Postgres command to promote a slave to the master role.

After promoting a slave, you need to reconfigure all the web servers to use the new master database. The easiest way to do that is through a deployment:

```
fab <environment> deploy_worker deploy_web
```

If you had more than one slave database before promoting a slave, the additional slaves need to be reset to stream from the new master. This can be accomplished with the `reset_slaves` command:

```
fab <environment> reset_slaves
```

Once complete, you can disable the upgrade message and resume usage of the site:

```
fab <environment> end_upgrade
```

Slave database goes down

If a slave database goes down, first enable the pretty upgrade message on all the web servers:

```
fab <environment> begin_upgrade
```

The site can operate in a degraded state with only a master database. To do that, navigate to the AWS console and stop or re-tag the old slave server so it can no longer be discovered by Fabric. Then, run a deployment to update the local settings files on all the web servers:

```
fab <environment> deploy_worker deploy_web
```

Once complete, you can disable the upgrade message and resume usage of the site:

```
fab <environment> end_upgrade
```

Adding a new slave

If a slave database is lost (either due to promotion to the master role or because it was itself lost), it is desirable to return the application to having two or more database servers as soon as possible. To add a new slave database to the Postgres cluster, first create a new server as follows:

```
fab new:myproject,<environment>,db-slave,X
```

where X is the availability zone in which you wish to create the server (it should be created in a zone that doesn't already have a database server, or has the fewest database servers).

Next, configure the web servers to begin using the new slave by doing a serial deployment:

```
fab deploy_serial:myproject,<environment>
```

This will take the web servers down one at a time, deploy the latest code, and update the settings file to use the newly added database server.

Slave database loses replication connection

While PostgreSQL administration is outside the scope of this guide, if you have determined that a slave database has lost the replication connection to the master database and you prefer not to simply create a new slave database server, you can re-sync the slave(s) with the master with the following command:

```
fab <environment> reset_slaves
```

Web server dies

Web servers are disposable, and are automatically recreated by via autoscaling if they become unhealthy.

Worker server dies

Worker servers are also disposable, so the easiest way to recover from one dying is simply to destroy it and create another. To destroy the instance, make sure that it's really dead (try SSHing to it and/or rebooting it from the AWS console). If all else fails, you can terminate the instance from the console (unless you want to leave it around to troubleshoot what went wrong).

Adding a new worker server

Creating a new worker server works the same as creating a web server:

```
fab new:myproject,<environment>,worker,X
```

where X is the availability zone in which you wish to create the server.

After creating the worker, you will also need to update it with correct settings file and start the worker processes. This can be done by running:

```
fab <environment> deploy_worker
```

Cache service goes down

If one of the services (e.g., RabbitMQ or Redis) simply dies on the cache server, SSH to that machine and attempt to start it by hand. RabbitMQ has been known on at least one occasion to have shutdown by itself for no apparent reason.

Cache server (RabbitMQ and Redis) fails

If the cache server fails, the web site will be inaccessible until a new server is created because the site relies on using Redis as a session store. As such, first display the pretty upgrade message on the servers:

```
fab <environment> begin_upgrade
```

Now, create a new cache server as follows:

```
fab new:myproject,<environment>,cache,X
```

where X is the availability zone in which you wish to create the server. Typically this should be one of the two zones that the web servers reside in.

While the new server is being created, navigate to the AWS console and stop or re-tag the old cache server so it can no longer be discovered by Fabric.

Once the new server has finished building, update the configuration on all the servers by running a deployment:

```
fab <environment> deploy_worker deploy_web
```

When that's complete, disable the upgrade message on the web servers:

```
fab <environment> end_upgrade
```

Web servers churning during a deploy

If you see web servers being launched, but then being terminated before they come into service, this is usually due to a problem with the load balancer not receiving a healthy response from the health check. If the web server is returning a 500 error, you should hopefully get an error email, which will help you debug the problem. If you get a 4xx error, you may not, so you might not even be aware that the web servers are churning. Once you are aware, suspend autoscaling:

```
fab suspend_autoscaling_processes:myproject,<environment>
```

SSH into the web server in question. Look at the `/home/myproject/www/{environment}/log/access.log` and see what HTTP status code is being returned to the load balancer.

- 401 errors mean the load balancer is getting a Basic Auth check which it is failing.
- 404 errors mean the health check URL is incorrectly configured, either due to a misconfiguration in Nginx or in Django.

Remember to resume autoscaling once you have fixed the problem:

```
fab resume_autoscaling_processes:myproject,<environment>
```

Using FabulAWS in your fab file

FabulAWS uses Fabric internally to communicate with newly-created servers, so it follows naturally that you can use FabulAWS in your fab files to create new servers and deploy code to them.

Simple Fabric example

Adding `fabulaws` to an existing fab file can be as simple as importing and instantiating an EC2 instance class, e.g.:

```
from fabric.api import *
from fabulaws.ec2 import MicroUbuntuInstance

def new_instance():
    i = MicroUbuntuInstance()
    env.hosts = [i.hostname]

def bootstrap():
    run('git clone %s' % env.repo)
```

The `new_instance` method creates a new Amazon EC2 instance and gives you access to the hostname of that newly created instance, so running:

```
fab new_instance bootstrap
Connecting to EC2...
```

would create a new copy of that instance on Amazon, using the API key in your shell environment.

Tagging instances

To make it easier to keep track of your instances on EC2, you can tag them with your environment (e.g., `'staging'` or `'production'`), as well as something that identifies the product or group of servers that you're deploying:

```
from fabric.api import *
from fabulaws.ec2 import MicroUbuntuInstance

def new_instance(environment):
    tags = {'environment': environment, 'product': 'caktus-website'}
    i = MicroUbuntuInstance(tags=tags)
    env.hosts = [i.hostname]

def bootstrap():
    run('git clone %s' % env.repo)
```

Now, you can pass the environment that you're creating into when you run `fab`:

```
fab new_instance:staging bootstrap
Connecting to EC2...
```

Retrieving tagged instances

To retrieve and use tagged instances from your fab file, use the `ec2_hostnames` method in `fabulaws.api` to retrieve the hostnames for the instances tagged with the appropriate tags, e.g.:

```
from fabric.api import *
from fabulaws.api import *

def staging():
    filters = {'tag:environment': 'staging', 'tag:product': 'caktus-website'}
    env.hosts = ec2_hostnames(filters=filters)

def update():
    run('git pull')
```

Then, you can run `fab` as you normally would from the command line, and it will reach out to EC2 to retrieve the hostname(s) for your server(s) before running commands on them:

```
$ fab staging deploy
Connecting to EC2...
```

Useful commands

Mega-commands

- `fab describe:deployment,environment` - Show the config and existing servers
- `fab create_environment:deployment,environment` - create all the initial servers. After this, you might need to bump up the instances in the autoscaling group to get web servers going.
- `fab update_environment:deployment,environment` - run update_sysadmin_users, update_server_passwords, and upgrade_packages
- `fab environment update_services` - does upload_newrelic_conf, upload_supervisor_conf, upload_pgbouncer_conf, and upload_nginx_conf

Deploys

All-in-one commands:

- `fab deploy_serial:deployment,environment[,launch_config_name]` - Create a new launch config if a name is not provided. Update the ASG to use the provided or new launch config. Take web servers down one at a time and bring up new ones, so you end up with all new ones without downtime.
- `fab deploy_full:deployment,environment[,launch_config_name]` - Create a new launch config if a name is not provided. Update the ASG to use the provided or new launch config. Take all the web servers down and bring up new ones. This is faster than deploy_serial but does cause downtime.

Note: Neither of the above commands updates secrets on the servers. Be sure to explicitly run `fab environment update_server_passwords` if you want to change secrets at the time of deployment.

More low-level commands:

- `fab create_launch_config_for_deployment:deployment,environment` - Create a new launch config and print its name, but do not use it for anything. Typically you could use this and then follow with one of the deploy commands, providing the launch_config_name that was output from this command.
- `fab environment begin_upgrade` - puts up a maintenance page for all requests (the deploy_xxx commands do this for you)
- `fab environment deploy_web[:changeset]` - deploy to web servers (update them in place, do not update LC or ASG), restart processes
- `fab environment deploy_worker[:changeset]` - deploy to worker (update in place), restart processes
- `fab environment flag_deployment` - sends a message to New Relic that the current code revision has just been deployed
- `fab environment end_upgrade` - reverses begin_upgrade

Misc

- `fab environment update_sysadmin_users` - create or update dev users on servers
- `fab environment upgrade_packages` - upgrade all Ubuntu packages on servers
- `fab environment mount_encrypted` - see source

EC2 instances

- `fab new:deployment,environment,role[,avail_zone[,count]]`

Handling secrets

- `fab environment update_local_fabsecrets`
- `fab environment update_server_passwords` - push secrets from local file to servers (except luks_passphrase)

Supervisor

- `fab environment upload_supervisor_conf`
- `fab environment supervisor:command,group[,process]` - runs 'supervisorctl command environment-group:environment-process', or 'supervisorctl command environment-group'

Examples:

- fab testing supervisor:stop,web
- fab testing supervisor:stop,celery
- fab testing supervisor:stop,pgbouncer
- fab testing supervisor:start,pgbouncer etc.

Python/Django

web servers & worker:

- `fab environment update_requirements` - does a pip install (without -U) (on all webs & worker)
- `fab environment update_local_settings` - render local settings template and install it on the servers (but does not restart services) (on all webs & worker)
- `fab environment bootstrap` - clones source repo, updates services, creates an virtual env and installs Python packages (on all webs & worker)
- `fab environment clone_repo` – clones the source repo (on all webs & worker)
- `fab environment update_source` - updates checked-out source (on all webs & worker)
- `fab environment current_changeset` - check latest code from repo (on all webs & worker)

worker only:

- `fab environment managepy:command` - run a manage.py command on worker
- `fab environment migrate` - run a migrate command on worker
- `fab environment collectstatic` - run a migrate command on worker
- `fab environment dbbackup` - run a database backup using dbbackup
- `fab environment dbrestore` - run a database restore - see code for now for more info

Databases

- `fab environment upload_pgbouncer_conf`
- `fab reload_production_db[:prod_env[,src_env]]`
- `fab reset_local_db:dbname`
- `fab environment reset_slaves` - this resets the config & data on the db slaves and is a good way to get things back into a working state if the replication seems broken
- `fab environment promote_slave[:index]` - change slave *index* to be the master. After this, run update_local_settings to make the web servers use the new settings.

Nginx

- `fab environment upload_nginx_conf`
- `fab environment restart_nginx`

Newrelic

- `fab environment upload_newrelic_conf`
- `fab update_newrelic_keys:deployment,environment` - especially useful because it restarts the Django processes, even if you don't need to change the New Relic config.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search