touchdown documentation

Release 0.15.14

John Carr

Jun 09, 2017

Contents

Touchdown is a tool for launching and managing infrastructure services - be they physical servers, virtual subnets or private dns records.

CHAPTER 1

Overview

You can use Touchdown to build and manage infrastructure safely and repeatably. It's python API can be used to describe the components of a simple hello world application or an entire datacenter.

Touchdown is infrastructure as code. Whether you want disposable and repeatable builds of a single application or to create a blueprint deployment strategy for users of your web framework you can now treat it as another versionable development artifact.

Using dependency information inferred from your configuration Touchdown can generate efficient plans for creating new environments, updating existing ones or tearing old ones down. You can see the changes it will make before it makes them.

Under the hood this dependency information actually forms a graph. This enables other features beyond just deploying configuration changes. Today we can visualize your infrastructure to help you see how your components are connected, but that is just the beginning.

Goals

Strict It is frustrating and potentially even dangerous to have an operation fail part way through. It can be costly if it fails part way through a slow operation. So we do as much validation as possible before evening making the first API call.

Declarative The aim is to describe your environment in a declarative way. What should your infrastructure look like, not what changes to make to it. The metadata this gives about your environment is then useful for more than just creating an instance.

Idempotent Being able to run the tool twice and be confident that you won't end up with 2 semi-broken instances. Idempotence means it is safe to apply a configuration multiple times.

Repeatable Once your Touchdown configuration is working you should be able to tear it down and rebuild it, and get something configured exactly the same.

Direction

The first phase of Touchdown is concentrating on building a solid foundation with good support of Amazon technologies - from low level compute instances up to the outward facing services like CloudFront.

CHAPTER 2

Installation

Installing from PyPi

You can install Touchdown from PyPi with pip. The suggested way is to install it in a virtualenv:

```
pip install touchdown
```

Right now we don't depend on optional libraries. In order to work with AWS you will need to install botocore:

```
pip install botocore
```

And in order to deploy server configuration you'll need to install fuselage:

```
pip install fuselage
```

Installing from GitHub

If you are hacking on Touchdown the recommended way to get started is to clone the repo and build a virtualenv:

```
git clone git://github.com/yaybu/touchdown
cd touchdown
virtualenv .
source bin/activate
pip install -e .
```

CHAPTER 3

touchdown

Installing touchdown into a virtualenv creates a `touchdown` command. It will load a configuration from a `Touchdownfile` in the current directory.

The `Touchdownfile` is a python source code file. A workspace object is predefined so you can just start defining resources:

```
# My first Touchdownfile

aws = workspace.add_aws(region='eu-west-1')
vpc = aws.add_vpc(name='my-first-vpc', cidr_block="10.10.10.0/24")
vpc.add_subnet(name='myfirst-subnet', cidr_block="10.10.10.0/25")
```

You can apply this simple configuration with `touchdown apply`:

```
$ touchdown apply
[100.00%] Building plan...
Generated a plan to update infrastructure configuration:

vpc 'my-first-vpc':
  * Creating vpc 'my-first-vpc'
  * Waiting for resource to exist
  * Display resource metadata
  * Set tags on resource my-first-vpc
      Name = my-first-vpc

subnet 'myfirst-subnet':
  * Creating subnet 'myfirst-subnet'
  * Waiting for resource to exist
  * Display resource metadata
  * Set tags on resource myfirst-subnet
      Name = myfirst-subnet

Do you want to continue? [Y/n] y
[40.00%] [worker3] [vpc 'my-first-vpc'] Creating vpc 'my-first-vpc'
[40.00%] [worker3] [vpc 'my-first-vpc'] Waiting for resource to exist
```

```
[40.00%] [worker3] [vpc 'my-first-vpc'] Resource metadata:
[40.00%] [worker3] [vpc 'my-first-vpc']     VpcId = vpc-72a31c17
[40.00%] [worker3] [vpc 'my-first-vpc'] Set tags on resource my-first-vpc
[40.00%] [worker3] [vpc 'my-first-vpc']     Name = my-first-vpc
[60.00%] [worker5] [subnet 'myfirst-subnet'] Creating subnet 'myfirst-subnet'
[60.00%] [worker5] [subnet 'myfirst-subnet'] Waiting for resource to exist
[60.00%] [worker5] [subnet 'myfirst-subnet'] Resource metadata:
[60.00%] [worker5] [subnet 'myfirst-subnet']     SubnetId = subnet-cf8f3a96
[60.00%] [worker5] [subnet 'myfirst-subnet'] Set tags on resource myfirst-subnet
[60.00%] [worker5] [subnet 'myfirst-subnet']     Name = myfirst-subnet
```

It's idempotent so you can run it again:

```
$ touchdown apply
[100.00%] Building plan...
Planning stage found no changes were required.
```

And you can tear it down with `touchdown destroy`:

```
$ touchdown destroy
[100.00%] Building plan...
Generated a plan to update infrastructure configuration:

subnet 'myfirst-subnet':
  * Destroy subnet 'myfirst-subnet'

vpc 'my-first-vpc':
  * Destroy vpc 'my-first-vpc'

Do you want to continue? [Y/n] y
[20.00%] [worker1] [subnet 'myfirst-subnet'] Destroy subnet 'myfirst-subnet'
[60.00%] [worker4] [vpc 'my-first-vpc'] Destroy vpc 'my-first-vpc'
```

It takes the following arguments:

`--serial`
: Force Touchdown to deploy a configuration in serial. By default touchdown applies configuration in parallel using a dependency graph inferred from your configuration.

  Unlike parallel mode, serial mode is deterministic.

`--debug`
: Turns on extra debug logging. This is quite verbose. For AWS configurations this will show you the API calls that are made.

There are a bunch of commands you can run against your Touchdown config:

Applying configuration changes

You can apply configuration changes with the `apply` command:

```
touchdown apply
```

This will build a plan of what it will create or update and ask you to confirm before applying it. If you run the same configuration again no changes should be made.

Tearing down infrastructure

You can tear down any infrastructure managed with Touchdown using the `destroy` command:

```
$ touchdown destroy
```

This will generate a plan of what it will teardown and then prompt you before doing so.

Getting signon urls

Some services support generating urls for granting secure temporary access to their admin interfaces. For example, you can generate an AWS federation URL for any IAM Role that you can assume. Touchdown exposes this via its `get-signin-url` command. For example, for an AWS *`Role`* defined like this:

```
aws.add_role(
    name="deployment",
    assume_role_policy={...},
    policies={...},
)
```

You can:

```
touchdown get-signin-url deployment
```

To get a url that allows you to see the AWS console with just the policies attached to that role.

Tailing logs

You can tail your logs with the `tail` command.

For example if you have a CloudWatch log group defined:

```
aws.add_log_group(
    name="application.log",
)
```

Then you could get the last 15 minutes of log events with:

```
touchdown tail application.log -s 15m
```

And you could stream the logs as they are ingested with:

```
touchdown tail application.log -f
```

You can use the following arguments:

`--start, -s`
: The time to start fetching logs from.

`--end, -e`
: The time to fetch logs until.

`--follow, -f`
: Don't exit. Continue to monitor the log stream for new events.

Rolling back your data

You can rollback your application state with the `rollback` command.

For example, if you have an Amazon RDS database called `foo` you can rollback the last 15m of changes with:

```
touchdown rollback foo 15m
```

Or you could revert it to a named snapshot with:

```
touchdown rollback foo mysnapshot
```

scp

If you have defined an explicit ssh connection in your config:

```
aws.add_auto_scaling_group(
    name=name,
    launch_configuration=...,
    <SNIP>,
)

workspace.add_ssh_connection(
    name="worker",
    instance="worker",
    username="ubuntu",
    private_key=open('foo.pem').read(),
)
```

Then you could scp files to and from it with:

```
touchdown scp foo.txt worker:
```

And in reverse:

```
touchdown scp worker:foo.txt /tmp/
```

You can use the following arguments:

Snapshotting your data

You can snapshot your database with the `snapshot` command.

For example, if you have an Amazon RDS database called `foo` you can create a snapshot with:

```
touchdown snapshot foo mysnapshot
```

You can then revert to it with:

```
touchdown rollback foo mysnapshot
```

SSH

If you have defined an explicit ssh connection in your config:

```
aws.add_auto_scaling_group(
    name=name,
    launch_configuration=...,
    <SNIP>,
)

workspace.add_ssh_connection(
    name="worker",
    instance="worker",
    username="ubuntu",
    private_key=open('foo.pem').read(),
)
```

Then you could ssh into a random instead in the `worker` autoscaling group with:

```
touchdown ssh worker
```

You can use the following arguments:

Generating graphs

You can generate a graph of your infrastructure with the `dot` command:

```
$ touchdown dot
```

This will output a `dot` file that can be processed with graphviz or displayed with a tool like xdot. On Ubuntu you can run this from the commandline:

$ touchdown dot > mygraph.dot $ xdot mygraph.dot

CHAPTER 4

Tutorials

Hello world: A static site with S3, Route53 and CloudFront

Here is a simple `Touchdownfile` that creates a bucket and sets up Route53 DNS for it:

```
aws = workspace.add_aws(
    access_key_id='AKI.....A',
    secret_access_key='dfsdfsdgrtjhwluy52i3u5ywjedhfkjshdlfjhlkwjhdf',
    region='eu-west-1',
)

bucket = aws.add_bucket(
    name="example.com",
)

hosted_zone = aws.add_hosted_zone(
    name="example.com",
    records=[{
        "type": "A",
        "alias": bucket,
    }]
)
```

All configurations start at the workspace object.

We ask the workspace for an AWS account for a given set of credentials and for a specific region.

To that AWS account we add a bucket to store our static website.

Then we add a Route53 zone. We pass in the bucket to the `alias` parameter. Alias records are a bit like server-side `CNAMES`. You can pass any resource to the `alias` parameter that has a hosted zone id. See the *`HostedZone`* documentation for the full list.

From this configuration Touchdown knows it must create a bucket before it can update the hosted zone. And it knows it must have perform any account setup steps before it can touch the bucket or hosted zone.

Handling S3 events with lambda functions

Suppose you store incoming media (such as .jpg or .png) in an incoming bucket and want to resize it into an output bucket. In this walkthrough we will use AWS Lambda to perform the transformation automatically - triggered by S3 `object-created` events.

> **Warning:** This example assumes 2 separate buckets are used. Attempting to use one bucket will result in recursion.

First of all we need a function to handle the images. In this example we just copy them straight to the destination bucket, but you can easily add your own transformation logic. Because this function is entirely self contained it can live in your *Touchdownfile*:

```
def resize_handler(event, context):
    import botocore.session
    session = botocore.session.get_session()
    s3 = session.create_client('s3', region_name='eu-west-1')
    for record in event['Records']:
        s3.copy_object(
            Bucket='resized',
            CopySource=record['s3']['bucket']['name'],
            Key=record['s3']['object']['key'],
        )
```

As it's an AWS example we need to setup an AWS workspace:

```
aws = workspace.add_aws(
    access_key_id='AKI.....A',
    secret_access_key='dfsdfsdgrtjhwluy52i3u5ywjedhfkjshdlfjhlkwjhdf',
    region='eu-west-1',
)
```

We need a role for lambda to use. These are the permissions that a lambda function will have. It **needs** access to push logs to CloudWatch logs. It needs access to read/write from our source and destination S3 buckets:

```
resize_role = aws.add_role(
    name="resize-role",
    policies={
        "logs": {
            "Statement": [{
                "Action": [
                    "logs:CreateLogGroup",
                    "logs:CreateLogStream",
                    "logs:PutLogEvents"
                ],
                "Effect": "Allow",
                "Resource": "arn:aws:logs:*:*:*"
            }]
        }
    },
    assume_role_policy={
        "Statement": [{
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.amazonaws.com"
            },
```

```
            "Action": "sts:AssumeRole"
        }],
    },
)
```

Then we can upload the actual lambda function. By default this will be a `python2.7` lambda function. Java and JavaScript can be uploaded as well, but you will need to set `runtime`. If you set `code` to a callable, Touchdown will automatically generate a zip to upload:

```
resize = aws.add_lambda_function(
    name="resize-media",
    role=resize_role,
    code=resize_handler,
    handler="main.resize_handler",
)
```

We need a source bucket, and we need to set up *notify_lambda* to invoke our lambda function whenever any of the `s3:ObjectCreated:*` events happen:

```
incoming = aws.add_bucket(
    name="incoming",
    notify_lambda=[{
        "name": "resize",
        "events": ["s3:ObjectCreated:*"],
        "function": resize,
    }],
)
```

And we need an output bucket:

```
resized = aws.add_bucket(
    name="resized",
)
```

Deploying Django at Amazon

We will deploy the sentry service at Amazon with Touchdown. Sentry is a Django application so much of this will be applicable to any Django application. This walkthrough will touch on:

- Creating a *VPC* with multiple interconnected *Subnet*'s.
- Creating a *Database* and passing its connection details to the Django instance.
- Using an *AutoScalingGroup* to start an instance.
- Using a *LoadBalancer* to scale up your service.

Desiging your network

We will create a subnet for each type of resource we plan to deploy. For our demo this means there will be 3 subnets:

segment	network	ingress
lb	`192.168.0.0/24`	`0.0.0.0/0:80 0.0.0.0/0:443`
app	`192.168.1.0/24`	`lb:80`
db	`192.168.2.0/24`	`app:5432`

The only tier that will have public facing IP's is the lb tier.

First we'll create a 3 subnet VPC:

```
vpc = aws.add_vpc('sentry')

subnets = {
    'lb': vpc.add_subnet(
        name="lb",
        cidr_block='192.168.0.0/24',
    ),
    'app': vpc.add_subnet(
        name="app",
        cidr_block='192.168.1.0/24',
    ),
    'db': vpc.add_subnet(
        name="db",
        cidr_block='192.168.2.0/24',
    ),
}
```

Then we'll create security groups that limit who can access the subnets:

```
security_groups = {}
security_groups['lb'] = vpc.add_security_group(
    name="lb",
    ingress=[
        {"port": 80, "network": "0.0.0.0/0"},
        {"port": 443, "network": "0.0.0.0/0"},
    ],
)
security_groups['app'] = vpc.add_security_group(
    name="app",
    ingress=[
        {"port": 80, "security_group": security_groups["lb"]},
    ],
)
security_groups['db'] = vpc.add_security_group(
    name="db",
    ingress=[
        {"port": 5432, "security_group": security_groups["app"]},
    ],
)
```

Adding a database

Rather than manually deploying postgres on an EC2 instance we'll use RDS to provision a managed *Database*:

```
database = aws.add_database(
    name=sentry,
    allocated_storage=10,
    instance_class='db.t1.micro',
    engine="postgres",
    db_name="sentry",
    master_username="sentry",
    master_password="password",
    backup_retention_period=8,
```

```
    auto_minor_version_upgrade=True,
    publically_accessible=False,
    storage_type="gp2",
    security_groups=[security_groups['db']],
    subnet_group=aws.add_db_subnet_group(
        name="sentry",
        subnets=subnets['db'],
    )
)
```

Building your base image

We'll setup a fuselage bundle to describe what to install on the base ec2 image:

```
provisioner = workspace.add_fuselage_bundle()
```

One unfortunate problem with Ubuntu 14.04 is that you can SSH into it before it is ready. `cloud-init` is still configuring it, and so if you start deploying straight away you will hit race conditions. So we'll wait for `cloud-init` to finish:

```
# Work around some horrid race condition where cloud-init hasn't finished running
# https://bugs.launchpad.net/cloud-init/+bug/1258113
provisioner.add_execute(
    command="python -c \"while not __import__('os').path.exists('/run/cloud-init/
↪result.json'): __import__('time').sleep(1)\"",
)
```

Then we'll install some standard python packages:

```
provisioner.add_package(name="python-virtualenv")
provisioner.add_package(name="python-dev")
provisioner.add_package(name="libpq-dev")
```

We are going to deploy the app into a virtualenv at `/app`. We'll do the deployment as root, and at runtime the app will use the *sentry* user. We'll create a `/app/etc` directory to keep settings in:

```
provisioner.add_group(name="django")

provisioner.add_user(
    name="django",
    group="django",
    home="/app",
    shell="/bin/false",
    system=True,
)

provisioner.add_directory(
    name='/app',
    owner='root',
    group='root',
)

provisioner.add_directory(
    name='/app/etc',
    owner='root',
    group='root',
```

```
)

provisioner.add_directory(
    name='/app/var',
    owner='root',
    group='root',
)

provisioner.add_execute(
    name="virtualenv",
    command="virtualenv /app",
    creates="/app/bin/pip",
    user="root",
)
```

We'll inject a requirements.txt and install sentry into the virtualenv:

```
provisioner.add_file(
    name='/app/requirements.txt',
    contents='\n'.join(
        'sentry==7.5.3',
    )
)

provisioner.add_execute(
    command="/app/bin/pip install -r /app/requirements.txt",
    watches=['/app/requirements.txt'],
)
```

This uses the *watches* syntax. This means we only update the virtualenv if requirements.txt has changed and is one mechanism for idempotence when using the `Execute` resource.

We need to actually start sentry. We'll use upstart for this:

```
provisioner.add_file(
    name="/etc/init/kickstart.conf",
    contents="\n".join([
       "start on runlevel [2345]",
       "task",
       "exec /app/bin/sentry kickstart",
    ]),
)
```

`kickstart` is a command we'll create that loads metadata such as the database username and password from AWS. It will use `initctl emit` to tell upstart other tasks it might need to start.

We'll also need upstart configuration for the django app server and for the celery processes:

```
provisioner.add_file(
    name="/etc/init/application.conf",
    contents="\n".join([
        "start on mode-application",
        "stop on runlevel [!2345]",
        "setuid sentry",
        "setgid sentry",
        "kill timeout 900",
        "respawn",
        " ".join([
```

```
            "exec /app/bin/gunicorn -b 0.0.0.0:8080",
            "--access-logfile -",
            "--error-logfile -",
            "--log-level DEBUG",
            "-w 8",
            "-t 120",
            "--graceful-timeout 120",
            "sentry.wsgi",
        ]),
    ]),
)

provisioner.add_file(
    name="/etc/init/worker.conf",
    contents="\n".join([
        "start on mode-worker",
        "stop on runlevel [!2345]",
        "setuid sentry",
        "setgid sentry",
        "kill timeout 900",
        "respawn",
        "exec /app/bin/django celery worker --concurrency 8",
    ]),
)

provisioner.add_file(
    name="/etc/init/beat.conf",
    contents="\n".join([
        "start on mode-beat",
        "stop on runlevel [!2345]",
        "setuid sentry",
        "setgid sentry",
        "kill timeout 900",
        "respawn",
        "exec /app/bin/django celery beat --pidfile=",
    ]),
)
```

To actually provision this as an AMI we use the *Image* resource:

```
image = aws.add_image(
    name="sentry-demo",
    source_ami='ami-d74437a0',
    username="ubuntu",
    provisioner=provisioner,
)
```

Deploying an instance

We'll deploy the image we just made with an auto scaling group. We are going to put a load balancer in front, which we'll set up first:

```
lb = aws.add_load_balancer(
    name='balancer',
    listeners=[
        {"port": 80, "protocol": "http", "instance_port": 8080, "instance_protocol":
↪"http"}
```

```
    ],
    subnets=subnets['lb'],
    security_groups=[security_groups['lb']],
    health_check={
        "interval": 30,
        "healthy_threshold": 3,
        "unhealthy_threshold": 5,
        "check": "HTTP:8080/__ping__",
        "timeout": 20,
    },
    attributes={
        "cross_zone_load_balancing": True,
        "connection_draining": 30,
    },
)
```

We are going to set some user data in the AutoScaling setup so that Django knows which database to connect to.

user_data = serializers.Json(serializers.Dict({

"DATABASES": serializers.Dict(ENGINE='django.db.backends.postgresql_psycopg2', NAME=database.db_name, HOST=serializers.Format("{0[Address]}", database.get_property("Endpoint")), USER=database.master_username, PASSWORD=database.master_password, PORT=5432,),

}))

Then we need a *LaunchConfiguration* that says what any started instances should look like and the *AutoScalingGroup* itself:

```
app = aws.add_auto_scaling_group(
    name="sentry-app",
    launch_configuration=aws.add_launch_configuration(
        name="sentry-app",
        image=ami,
        instance_type="t1.micro",
        user_data=user_data,
        key_pair=keypair,
        security_groups=security_groups["app"],
        associate_public_ip_address=False,
    ),
    min_size=1,
    max_size=1,
    load_balancers=[lb],
    subnets=subnets["app"],
)
```

Creating an Amazon API Gateway for domain redirect

Eric Hammond published a blog post about getting started with API Gateway. He built a simple gateway that could redirect a vanity domain name. In this walkthrough we'll replicate that setup with Touchdown.

In the blog post there are some variables - lets replicate them:

```
base_domain = 'erichammond.xyz'
target_url = 'https://twitter.com/esh'
```

```
api_name = base_domain
api_description = "Redirect $base_domain to $target_url"
resource_path = "/"
stage_name = "prod"
region = "us-east-1"

certificate_name = base_domain
certificate_body = base_domain + ".crt"
certificate_private_key = base_domain + ".key"
certificate_chain = base_domain + "-chain.crt"
```

As it's an AWS example we need to setup an AWS workspace:

```
aws = workspace.add_aws(
    access_key_id='AKI.....A',
    secret_access_key='dfsdfsdgrtjhwluy52i3u5ywjedhfkjshdlfjhlkwjhdf',
    region='eu-west-1',
)

api = aws.add_rest_api(
    name=api_name,
    description=api_description,
)

root = api.get_resource(name='/')

root.add_method(
    method="GET",
    authorization_type = "NONE",
    api_key_required=False,
)

root.add_method_response(
    method="GET",
    status_code=301,
    response_models={"application/json": "Empty"},
    response_parameters={"method.response.header.Location": True},
)

root.add_integration(
    method="GET",
    type="MOCK",
    request_templates={
        "application/json": "{\"statusCode\": 301}",
    }
)

root.add_integration_response(
    method="GET",
    status_code=301,
    response_templates='{"application/json":" redirect"}',
    response_parameters={
        "method.response.header.Location": "'"'$target_url'"'",
    },
)
```

Deploying the API

```
deployment = api.add_deployment(
    name=base_domain,
    stage_name=stage_name,
    stage_description=stage_name,
)
stage = deployment.get_stage(name=stage_name)
```

Linking to a domain name

```
stage.add_domain(aws.add_apigateway_domain(
    domain=base_domain,
    certificate_name=certificate_name,
    certificate_body=ceritificate_body,
    certifcate_private_key=certifificate_private_key,
    certificate_chain=certificate_chain,
))
```

CHAPTER 5

Defining configuration

When using touchdown as a standalone tool then your configuration should be defined in the `Touchdownfile`. A `Touchdownfile` is a python file. The `workspace` variable will have been initialised for you so you can start connecting the components in your infrastructure. For example, to create a new VPC at Amazon with a subnet your `Touchdownfile` would contain:

```
aws = workspace.add_aws(
    access_key_id='....',
    secret_access_key='....',
    region='eu-west-1',
)

vpc = aws.add_vpc(
    name='my-vpc,
    cidr_block='192.168.0.0/24',
)

vpc.add_subnet(name='subnet1', cidr_block='192.168.0.0/25')
```

Amazon Web Services

Amazon Certificate Manager

Amazon Certificate Manager generates free certificates for TLS with Elastic Load Balancer and CloudFront, and transparently handles rotation and renewal.

When you request a certificate Amazon validate you control the domain by e-mail. For example if you requested a certificate for `www.example.com` it attempts to contact:

- The domain registrant
- The technical contact
- The administrative contact

- admin@www.example.com
- administrator@www.example.com
- hostmaster@www.example.com
- postmaster@www.example.com
- webmaster@www.example.com

Note: These certificates can only be used with Amazon services - there is no way to obtain the private certificate.

If you already have a certificate that you wish to use with CloudFront or ELB you can upload it with a `ServerCertificate`.

Creating a certificate

class `Certificate`

To create a certificate you just need to choose the domain it is for:

```
certificate = aws.add_acm_certificate(
    name='www.example.com',
)
```

`name`

The domain name to request a certificate for.

`validation_options`

By default ACM will e-mail the contacts for your domain - so *hostmaster@www.example.com* in the previous example. You can override this:

```
certificate = aws.add_acm_certificate(
    name="www.example.com",
    validation_options=[{
        "domain": "www.example.com",
        "validation_domain": "example.com",
    }]
)
```

`alternate_names`

A list of alternative domain names this cert should be valid for, for example for `www.example.com` you might also add `www.example.net`.

API Gateway

Amazon API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. With a few clicks in the AWS Management Console, you can create an API that acts as a "front door" for applications to access data, business logic, or functionality from your back-end services, such as workloads running on Amazon Elastic Compute Cloud (Amazon EC2), code running on AWS Lambda, or any Web application. Amazon API Gateway handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls, including traffic management, authorization and access control, monitoring, and API version management. Amazon API Gateway has no minimum fees or startup costs. You pay only for the API calls you receive and the amount of data transferred out.

Setting up a REST API

class `RestApi`

To start building an API you need to create a REST API component:

```
rest_api = aws.add_rest_api(
    name='my-api',
    description='....',
)
```

`name`

A name for this API.

`description`

A description for this API.

Defining resources

class `Resource`

`name`

The name of the resource. This is a uri, for example `/animal`.

There will be an implict `/` resource created, which you can attach other resources to:

```
resource = rest_api.get_resource(name='/')
animal = resource.add_resource(
    name='/animal',
)
```

`parent_resource`

The resource this resource is attached to:

```
dog = rest_api.add_resource(
    name='/animal/dog',
    parent_resource=animal,
)
```

This is optional if you attach a resource directly:

```
dog = animal.add_resource(name='/animal/dog')
```

Defining models

class `Model`

```
rest_api.add_model(
    name='dog',
    description='dog schema',
    schema='',
    content_type='application/json',
)
```

`name`

description

schema

content_type
This defaults to `application/json`.

Defining deployments

class Deployment

```
rest_api.add_deployment(
    name='api-deployment',
    stage='production',
)
```

name

stage

stage_description

cache_cluster_enabled

cache_cluster_size

variables

Adding stages

A stage defines the path through which an API deployment is accessible. With deployment stages, you can have multiple releases for each API, such as alpha, beta, and production. Using stage variables you can configure an API deployment stage to interact with different backend endpoints.

class Stage
You attach new stages to a deployment:

```
my_stage = deployment.add_stage(
    name='staging',
)
```

name

description

cache_cluster_enabled

cache_cluster_size

variables

Attaching methods

class Method
You attach an method to a resource:

```
my_method = resource.add_method(
    method = "GET",
)
```

name

authorization_type

api_key_required

request_parameters

request_models

Attaching method responses

class MethodResponse

You attach an method response to a resource:

```
my_method_response = resource.add_method_response(
    name = "GET",
)
```

name

status_code

response_parameters

response_models

Attaching integrations

class Integration

You attach an integration to a resource:

```
my_integration = resource.add_integration(
    name = "GET",
)
```

name
E.g. `GET`

integration_type
Can be *HTTP*, *AWS* or *MOCK*.

integration_http_method

request_parameters

request_templates

uri

credentials

cache_namespace

cache_key_parameters

Attaching integration responses

class IntegrationResponse

You attach an integration response to a resource:

```
my_integration_response = resource.add_integration_response(
    name = "GET",
)
```

name

E.g. `GET`

status_code

selection_pattern

response_parameters

response_templates

Authentication

Access to AWS services is authenticated used a pair of credentials called the `access_key_id` and the `secret_access_key`. A single user account can have multiple access keys associated with it, and via the STS service you can generate access keys directly for a role (rather than for a user).

Access keys

The simplest way to start performing actions against AWS is to add a `Account` object to your workspace:

```
aws = workspace.add_aws(
    access_key_id='AKIDFKJDKFJF',
    secret_access_key='skdfkoeJIJE4e2SFF',
    region='eu-west-1',
)
```

If you will be orchestrating AWS services from within AWS you can use a *touchdown.aws.iam.InstanceProfile* to grant temporary credentials to an EC2 instance. Touchdown will automatically retrieve them from the AWS metadata service when you don't specify an `access_key_id`:

aws = workspace.add_aws(region='eu-west-1',

)

Assuming a role

If you have multiple accounts at Amazon (perhaps one per customer) and have a shared resource - such as a Route53 zone - then you can use cross-account roles to manage it.

In the account with the shared resource you can create a role as follows:

```
aws.add_role(
    name="route53_full_access_{}".format(env.environment),
    assume_role_policy={
        "Statement": [{
            "Effect": "Allow",
            "Principal": {"AWS": "arn:aws:iam::{}:root".format(env.account)},
```

```
            "Action": "sts:AssumeRole"
        }]
    },
    policies={
        "route53_full_access": {
            "Statement": [{
                "Effect": "Allow",
                "Action": ["route53:*"],
                "Resource": ["*"]
            }]
        }
    },
)
```

Now in your other account you can assume this role:

```
other_account = aws.add_external_role(
    name='my-role',
    arn='',
)

other_account.add_hosted_zone(
    name='www.example.com',
)
```

Autoscaling

Why should I use autoscaling?

An *AutoScalingGroup* provides 2 kinds of automation:

- Dynamic scaling in response to CloudWatch metrics. For example you can monitor the length of a queue and start extra workers if the queue is growing instead of declining.
- Scheduled (time based) capacity changes

These are optional of course. You just manually manage the `desired_capacity` of a group to scale your application as you see fit.

Even if you are not using the scaling facilities of an autoscaling group there is still a strong reason to use them. By creating a *AutoScalingGroup* with `min` set to `1` and `max` set to `1` you ensure that AWS will try to replace any instance that has failed. If the instance goes down it will be replaced by a new one as defined by your launch configuration.

Setting up base autoscaling

class AutoScalingGroup

name
A name for this AutoScalingGroup. This field is required. It must be unique within an AWS account

subnets
A list of *Subnet* resources

launch_configuration
A *LaunchConfiguration*.

max_size
The maximum number of EC2 instances that can be started by this AutoScalingGroup.

min_size
The minimum number of EC2 instances that must be running

desired_capacity
The number of EC2 instances that should be running. Must be between min_size and max_size.

default_cooldown
The amount of time (in seconds) between scaling activities.

health_check_type
The kind of health check to use to detect unhealthy instances. By default if you are using ELB with the ASG it will use the same health checks as ELB.

load_balancers
A list of *LoadBalancer* resources. As instances are created by the auto scaling group they are added to these load balancers.

Defining what to launch

class LaunchConfiguration

name
A name for this LaunchConfiguration. This field is required. It must be unique within an AWS account

image

key_pair
A *KeyPair*. This is the public key that gets injected to new ec2 instances created by this launch configuration.

security_groups
A list of SecurityGroup.

user_data

instance_type

kernel

ramdisk

block_devices
This is not supported yet.

instance_monitoring

spot_price

instance_profile
A *InstanceProfile*. Use this to grant started instances a pair of ephemeral credentials for using other AWS services, such as S3.

ebs_optimized

associate_public_ip_address

placement_tenancy

Dynamic scaling based on CloudWatch

In this example we use a metric that will be populated by our application. It contains the length of a task queue:

```
queue1_length = aws.add_metric(
    name='queue1',
    namespace="Statsd/queue",
)
```

We've also got an autoscaling group. This is a pool of workers that we want to dynamically scale:

```
worker = aws.add_auto_scaling_group(
    name='worker',
    min=1,
    max=4,
    launch_configuration=<snip>,
)
```

We connect these together with an alarm and an autoscaling policy that will scale the worker pool up:

```
queue1_length.add_alarm(
    name='scaling-queue1-too-busy',
    statistic='Average',
    period=60,
    evaluation_periods=5,
    threshold=10,
    comparison_operator='GreaterThanOrEqualToThreshold',
    alarm_actions=[worker.add_policy(
        name='scale-up',
        adjustment_type='ChangeInCapacity',
        scaling_adjustment=1,
        cooldown=2 * 60,
    )],
)
```

And then scale the pool back down:

```
queue1_length.add_alarm(
    name='scaling-queue1-too-quiet',
    statistic='Average',
    period=60,
    evaluation_periods=5,
    threshold=0,
    comparison_operator='LessThanOrEqualToThreshold',
    alarm_actions=[worker.add_policy(
        name='scale-down',
        adjustment_type='ChangeInCapacity',
        scaling_adjustment=-1,
        cooldown=10 * 60,
    )],
)
```

class AutoScalingPolicy

name
A name for this policy. This field is required.

auto_scaling_group
The *AutoScalingGroup* to apply this policy to.

adjustment_type
The adjustment type. Valid values are:

ChangeInCapacity: Increases or decreases the existing capacity. For example, the current capacity of your Auto Scaling group is set to three instances, and you then create a scaling policy on your Auto Scaling group, specify the type as `ChangeInCapacity`, and the adjustment as five. When the policy is executed, Auto Scaling adds five more instances to your Auto Scaling group. You then have eight running instances in your Auto Scaling group: current capacity (3) plus ChangeInCapacity (5) equals 8.

ExactCapacity: Changes the current capacity to the specified value. For example, if the current capacity is 5 instances and you create a scaling policy on your Auto Scaling group, specify the type as ExactCapacity and the adjustment as 3. When the policy is executed, your Auto Scaling group has three running instances.

PercentChangeInCapacity: Increases or decreases the capacity by a percentage. For example, if the current capacity is 10 instances and you create a scaling policy on your Auto Scaling group, specify the type as PercentChangeInCapacity, and the adjustment as 10. When the policy is executed, your Auto Scaling group has eleven running instances because 10 percent of 10 instances is 1 instance, and 10 instances plus 1 instance is 11 instances.

min_adjustment_step
Used with `adjustment_type` with the value `PercentChangeInCapacity`, the scaling policy changes the `desired_capacity` of the Auto Scaling group by at least the number of instances specified in the value.

scaling_adjustment
The number by which to scale. `adjustment_type` determines the interpretation of this number (for example, as an absolute number or as a percentage of the existing group size). A positive increment adds to the current capacity and a negative value removes from the current capacity.

cooldown
The amount of time, in seconds, after a scaling activity completes and before the next scaling activity can start.

CloudFront

There are 2 kinds of CloudFront distribution:

- A 'Web' distribution that acts as a CDN for HTTP and HTTPS traffic
- A 'Streaming' distribution that acts as a CDN for RTMP traffic

Serving content over HTTP and HTTPS

Web distributions act an "origin pull" based content delivery network. This means they work a bit like a caching proxy like varnish.

There are several pieces that need configuring. Together these pieces are called a Distribution Config. They are:

- How should the distribution listen for traffic. What ports, what certs, what domains.
- What backend servers can traffic be sent to. These are origins.

- How should traffic be mapped from a request to an origin. For example, you might have a application cluster at `/` and a search cluster at `/search`. These are called cache behaviours, and can also change how aggressively you cache based on the URL.

Note: CloudFront configuration changes are slow

Any configuration changes to a distribution are slow - taking around 15 minutes. If using blue/green type techniques during deployment it is best to not do that switch at the CloudFront tier of your stack.

---

class `Distribution`

The minimum distribution is:

```
distribution = self.aws.add_distribution(
    name='www.example.com',
    origins=[{
        "name": "www",
        "domain_name": "backend.example.com",
    }],
    default_cache_behavior={
        "target_origin": "www",
    },
)
```

`name`
The name of the distribution. This should be the primary domain that it responds to.

`comment`
Any comments you want to include about the distribution.

`aliases`
Alternative domain names that the distribution should respond to.

`root_object`
The default URL to serve when the users hits the root URL. For example if you want to serve index.html when the user hits www.yoursite.com then set this to '/index.html'. The default is '/'

`enabled`
Whether or not this distribution is active. A distribution must be enabled before it can be accessed by a client. It must be disabled before it can be deleted.

`origins`
A list of `Origin` resources that the Distribution acts as a front-end for.

`default_cache_behavior`
How the proxy should behave when none of the rules in `behaviors` have been applied.

`behaviors`
A list of *`CacheBehavior`* rules about how to map incoming requests to `origins`.

`error_responses`
A list of *`ErrorResponse`* rules that customize the content that is served for various error conditions.

`logging`
A *`LoggingConfig`* resource that describes how CloudFront should log.

`price_class`
The price class. By default `PriceClass_100` is used, which is the cheapest.

`ssl_certificate`
A *`ServerCertificate`*.

ssl_support_method
If this is set to `sni-only` then CloudFront uses the SNI mechanism. This only works on browsers newer than IE6. If you need maximum compatibility set it to `vip`. Your distribution will be assigned its own dedicated IP addresses, negating the need to use SNI. However, this is much more expensive.

ssl_minimum_protocol_version
The default value is `TLSv1`. To decrease the security of your system you can instead set this to `SSLv3`. **This is strongly discouraged**.

Serving content from an S3 bucket

You can pass a *S3Origin* to a CloudFront distribution to have it serve content from an S3 bucket. If you have a bucket called `my-test-bucket` then this looks like:

```
bucket = aws.add_bucket(name="my-test-bucket")

distribution = self.aws.add_distribution(
    name='www.example.com',
    origins=[{
        "name": "www",
        "bucket": bucket,
    }],
    default_cache_behavior={
        "target_origin": "www",
    },
)
```

You cannot use SSL for an S3 bucket backend - even if using HTTPS between the client and CloudFront, the connection between CloudFront and S3 will always be over unencrypted HTTP.

class S3Origin

name
A name for this backend service. This is used when defining cache behaviors.

bucket
A *Bucket* to serve content from.

origin_access_identity

Serving content from a backend HTTP or HTTPS service

CloudFront can act as a proxy for any HTTP or HTTP service. Just pass a *CustomOrigin* to a CloudFront distribution. For example, to serve content from `backend.example.com` on port 8080 abd 8443:

```
distribution = self.aws.add_distribution(
    name='www.example.com',
    origins=[{
        "name": "www",
        "domain_name": "backend.example.com",
        "http_port": 8080,
        "https_port": 8043,
    }],
    default_cache_behavior={
        "target_origin": "www",
```

```
    },
)
```

class CustomOrigin

name
A name for this backend service. This is used when defining cache behaviors.

domain_name
A backend server to contact.

http_port
The port that is serving HTTP content. The default value is `80`.

https_port
The port that is serving HTTPS content. The default value is `443`.

protocol
Specifies what protocol is used to contact this origin server. The default is `match-viewer`. This means that the backend is contacted with TLS if your client is using https. A less secure option is `http-only` which can be used to send even secure and confidential traffic in the clear to your backend.

ssl_policy
Specifies the permitted backend ssl versions. Defaults to SSLv3 and TLSv1.

You can also directly connect to an elb load balancer:

```
.. attribute:: load_balancer

    A `touchdown.aws.elb.LoadBalancer` instance to send HTTP or HTTP
    traffic to.
```

Cache behaviours

Particularly if you are using CloudFront in front of your entire site you might want different caching policies from different URL's. For example, there is no need to pass the query string or any cookies to the part of your site that serves CSS. This helps to improve cacheability.

class CacheBehavior

target_origin
The name of a *S3Origin* or *CustomOrigin* that this behaviour applies to.

forward_query_string
Whether or not to forward the query string to the origin server.

forward_headers
A whitelist of HTTP headers to forward to the origin server.

If you want to forward all headers you can set this to `['*']`. If you set it to an empty list no headers will be sent.

forward_cookies
A list of cookies to forward to the origin server.

If you want to forward all cookies you can set this to `['*']`. If you set it to an empty list no cookies will be sent.

viewer_protocol_policy
If set to `https-only` then all traffic will be forced to use TLS. If set to `redirect-to-https` then all HTTP traffic will be redirected to the https version of the url. `allow-all` passes on traffic to the origin using the same protocol as the client used.

default_ttl

min_ttl
The minimum amount of time to cache content for.

max_ttl

compress

allowed_methods
The HTTP methods that are passed to the backend.

cached_methods
The HTTP methods that might be cached. For example, it's unlikely that you would ever cache a `POST` request.

smooth_streaming
Whether or not to turn on smooth streaming.

Error handling

class ErrorResponse

error_code
A HTTP error code to replace with static content. For example, `503`.

response_page_path
A page to serve from your domain when this error occurs. If `/` was served by your application and `/static` was served from S3 then you would want to serve the page from `/static`, otherwise it is likely your error page would go down when your site went down.

response_code
By default this is the same as the `error_code`. However you can transform it to a completely different HTTP status code - even `200`!

min_ttl
How long can this error be cached for? It can be useful to set this to a low number for very busy sites - as it can act as a pressure release valve. However it is safest to set it to 0.

Access logging

class LoggingConfig
CloudFront can log some information about clients hitting the CDN and sync those logs to an S3 bucket periodically.

enabled
By default this is `False`. Set it to `True` to get CDN logs.

include_cookies
Set to `True` to include cookie information in the logs.

bucket
A *Bucket*.

path
A path within the S3 bucket to store the incoming logs.

Serving media over RTMP

A streaming distribution allows you to serve static media to your visitors over RTMP. You will need to serve the media player over HTTP(S) so you will probably use a streaming distribution in conjunction with a standard CloudFront distribution.

RTMP requests are accepted on ports 1935 and port 80. This is not configurable.

CloudFront supports:

- RTMP
- RTMPT (RTMP over HTTP)
- RTMPE (Encrypted RTMP)
- RTMPTE (Encrypted RTMP over HTTP)

class StreamingDistribution

name
The name of the streaming distribution. This should be the primary domain that it responds to.

comment
Any comments you want to include about the distribution.

aliases
Alternative names that the distribution should respond to.

enabled
Whether or not this distribution is active.

origin
A *S3Origin* that describes where to stream media from.

logging
A *StreamingLoggingConfig* resource that describes how CloudFront should log.

price_class
The price class. By default `PriceClass_100` is used, which is the cheapest.

class StreamingLoggingConfig

enabled
By default this is `False`. Set it to `True` to get CDN logs.

bucket
A *Bucket*.

path
A path within the S3 bucket to store the incoming logs.

Cloudtrail

Cloudtrail is the AWS audit log. It allows you to see what API calls a user has made (including the API calls generated by the AWS console).

Trail

class `Trail`

`name`
The name of the trail.

`bucket`
An S3 bucket used to store cloudtrail logs.

`bucket_prefix`

`topic`
Specifies a *Topic* defined for notification of log file delivery.

`include_global`
Specifies whether the trail is publishing events from global services such as IAM to the log files.

`cwlogs_group`
Specifies a CloudWatch logs group to deliver CloudTrail logs to.

`cwlogs_role`
Specifies a Role for the CloudWatch Logs endpoint to assume when writing to a LogGroup.

Cloudwatch

EC2 is the main workhorse of an AWS solution. It allows you to (manually or automatically) start virtual machines to run your application code.

Note: We recommend that your EC2 machines are stateless.

Metric

class `Metric`
This is a value that is tracked in the AWS CloudWatch service.

`name`
The name of the metric.

`namespace`

Alarm

class `Alarm`

`name`
Required. The name of the alarm. It must be unique within the account.

`description`
A human readable description of the alarm. May be up to 255 characters.

`actions_enabled`
If set to `True` then the actions defined will be executed when the alarm changes state.

ok_actions
A list of resources to notify when the alarm enters the `OK` state. Must be one of the following types:

- *Queue*
- Policy

alarm_actions
A list of resources to notify when the alarm enters the `ALARM` state. Must be one of the following types:

- *Queue*
- Policy

insufficient_data_actions
A list of resources to notify when the alarm enters the `INSUFFICIENT_DATA` state. Must be one of the following types:

- *Queue*
- Policy

metric
The metric this alarm is to respond to.

dimensions
Up to 10 dimensions for the associated metric. Use this to restrict the metric to a particular ec2 instance id or load balancer id.

statistic
The statistic to apply to the associated metric. Must be one of:

- SampleCount
- Average
- Sum
- Minimum
- Maximum

period
The period in seconds over which the specified statistic is applied.

unit
The unit for the alarm's associated metric. If specified, must be one of:

- Seconds
- Microseconds
- Milliseconds
- Bytes
- Kilobytes
- Megabytes
- Gigabytes
- Terabytes
- Bits
- Kilobits

- Megabits
- Gigabits
- Terabits
- Percent
- Count
- Bytes/Second
- Kilobytes/Second
- Megabytes/Second
- Gigabytes/Second
- Terabytes/Second
- Bits/Second
- Kilobits/Second
- Megabits/Second
- Gigabits/Second
- Terabits/Second
- Count/Second
- None

evaluation_periods
The number of periods over which data is compared to the specified threshold.

threshold
The value against which the specified statistic is compared.

comparison_operator
The operation to use when comparing `statistic` and `threshold`. For example, to dest when the statistic is less than threshold:

```
aws.add_alarm(
    name='myalarm',
    statistic='Average',
    threshold=5,
    comparison_operator='LessThanThreshold',
)
```

Must be one of:

- GreaterThanOrEqualToThreshold
- GreaterThanThreshold
- LessThanThreshold
- LessThanOrEqualToThreshold

Elastic Compute Cloud

EC2 is the main workhorse of an AWS solution. It allows you to (manually or automatically) start virtual machines to run your application code.

Note: We recommend that your EC2 machines are stateless.

Machine Instances

class Instance

You can add an EC2 instance with:

```
aws.add_ec2_instance(
    name='my-ec2-instance',
    ami='ami-cbb5d5b8',
)
```

name

ami

instance_type

key_pair

instance_profile

block_devices

subnet

security_groups

network_interfaces

tags

Additional storage

You can create EBS volumes to attach to your EC2 instance.

class Volume

You can add an EC2 volume with:

```
aws.add_volume(
    name='my-ec2-instance',
    availability_zone='eu-west-1a',
)
```

name

size

Size of the requested volume in GiB. Must be between 1 and 16384.

availability_zone

volume_type

iops

key

Machine Images

class Image
This represents a virtual machine image that can be used to boot an EC2 instance.

name

description

source_ami
An AMI to base the new AMI on.

username
The username to use when sshing to a new images.

steps
A list of steps to perform on the booted machine.

launch_permissions

tags

Key Pair

class KeyPair
In order to securely use SSH with an EC2 instance (whether created directly or via a AutoScalingGroup) you must first upload the key to the EC2 key pairs database. The KeyPair resource imports and keeps up to date an ssh public key.

It can be used with any AWS account resource:

```
aws.add_keypair(
    name="my-keypair",
    public_key=open(os.expanduser('~/.ssh/id_rsa.pub')),
)
```

name
The name of the key. This field is required.

public_key
The public key material, in PEM form. Must be supplied in order to upload a key pair.

ElastiCache

The ElastiCache service provides hosted REDIS and Memcache, with support for read replicas and high availability.

CacheCluster

class CacheCluster

name

instance_class
The kind of hardware to use, for example `db.t1.micro`

engine
The type of database to use, for example `redis` or `memcache`.

engine_version
The version of the cache engine to run

port
The TCP/IP port to listen on.

security_groups
A list of `SecurityGroup` to apply to this instance.

availability_zone
The preferred availability zone to start this CacheCluster in

multi_az
Whether or not to enable mutli-availability-zone features. This setting only applies when `engine` is `memcache`.

auto_minor_version_upgrade
Automatically deploy cache minor server upgrades

num_cache_nodes
The number of nodes to run in this cache cluster

subnet_group
A *SubnetGroup* that describes the subnets to start the cache cluster in.

parameter_group

apply_immediately

ReplicationGroup

class ReplicationGroup

name

description

primary_cluster
A *CacheCluster* resource.

automatic_failover

num_cache_clusters

instance_class
The kind of hardware to use, for example `db.t1.micro`

engine
The type of database to use, for example `redis`

engine_version
The version of the cache engine to run

port
The TCP/IP port to listen on.

security_groups
A list of `SecurityGroup` to apply to this instance.

availability_zone
The preferred availability zone to start this CacheCluster in

multi_az
Whether or not to enable mutli-availability-zone features

auto_minor_version_upgrade
Automatically deploy cache minor server upgrades

num_cache_nodes
The number of nodes to run in this cache cluster

subnet_group
A *SubnetGroup* that describes the subnets to start the cache cluster in.

parameter_group

apply_immediately

SubnetGroup

class SubnetGroup

name

subnets
A list of `Subnet` resources.

Elastic Transcoder

Pipeline

class Pipeline
..attribute:: name

The name of the pipeline. This field is required.

..attribute:: input_bucket

A *Bucket*.

..attribute:: output_bucket

A *Bucket*.

..attribute:: role = argument.Resource(Role, field="Role")

A *Role*.

..attribute:: key

A KMS key. Not currently supported.

..attribute:: notifications

An SNS notification topic. Not currently supported.

..attribute:: content_config

..attribute:: thumbnail_config

Elastic Load Balancer

Load Balancer

class LoadBalancer

name
The name of your load balancer. This is required.

listeners
A list of *Listener* resources. Determines what ports the load balancer should listen on and where traffic for those ports should be directed. You can only set a single backend port. All your application servers should be listening on the same port, not on ephemeral ports.

subnets
A list of `Subnet` resources. These are the subnets that the load balancer can create listeners in.

availability_zones
A list of availability zones this load balancer can listen in. If you set `subnets` then this option is implied and can be left unset.

scheme
By default this is `private`. This means the database is created on private ip addresses and cannot be accessed directly from the internet. It can be set to internet-facing if you want it to have a public ip address.

security_groups
A list of `SecurityGroup` resources. These determine which resources the LoadBalancer can access. For example, you could have a load balancer security group that only allowed access to your application instances, but not your database servers.

health_check
A *HealthCheck* instance that describes how the load balancer should determine the health of its members.

idle_timeout

connection_draining

cross_zone_load_balacning

access_log
An *Bucket* for storing access logs in.

Listeners

class Listener

protocol
The protocol to listen for. The choices are `HTTP`, `HTTPS`, `TCP` or `TCPS`.

port
A tcp/ip port to listen on.

instance_protocol
The protocol that your backend expects.

instance_port
The port that your backend is listening on.

ssl_certificate
This is a `ServiceCertificate`. This is required if your listener is over SSL.

Health checks

class HealthCheck

interval

check

healthy_threshold

unhealthy_threshold

timeout

Identity & Access Management

PasswordPolicy

class PasswordPolicy
You can set password policy on an Amazon account resource:

```
aws.add_password_policy(
    min_password_length=16,
)
```

min_password_length

require_symbols

require_numbers

require_uppercase

require_lowercase

allow_users_to_change_password

expire_passwords

max_password_age

password_reuse_prevention
Must be between 1 and 24.

hard_expiry

InstanceProfile

class InstanceProfile
You can create an `InstanceProfile` from an amazon account resource:

```
instance_profile = aws.add_instance_profile(
    name="my-instance-profile",
    roles=[my_role],
)
```

name

path

roles
A list of *Role* resources.

Role

class Role
You can create a Role from an amazon account resource:

```
role = aws.add_role(
    name="my-role",
    policies = {
        "s3-access": {
            # ... IAM policy definition ...
        }
    }
)
```

name

path

assume_role_policy
This field is a policy that describes who or what can assume this role. For example, if this is a role for EC2 instances you could set it to:

```
aws.add_role(
    name="my-role"
    assume_role_policy={
        "Statement": [{
            "Effect": "Allow",
            "Principal": {"Service": ["ec2.amazonaws.com"]},
            "Action": ["sts:AssumeRole"],
        }],
    },
)
```

policies
A dictionary of policies that apply when assuming this role.

ServerCertificate

class ServerCertificate
In order to use SSL with a *touchdown.aws.cloudfront.Distribution* or a *touchdown.aws.elb.LoadBalancer* you'll first need to upload the SSL certificate to IAM with the ServerCertificate resource.

name

path

certificate_body

certificate_chain

private_key

Key management service

The Key Management Service is a scaled and highly available API for managing encryption keys.

It is integrated with:

- RDS
- S3
- EBS
- Redshift
- EMR
- Elastic Transcoder
- WorkMail

With KMS you can create keys that can never be exported from the service and restrict encryption and decryption by IAM policy.

class Key
A key in the Amazon KMS service.

name
The description of the key. Must be at most 8192 characters.

..warning:: A key cannot be directly named.

Without a name there would be no way for touchdown to remember which key it created previously (without out-of-band state). In order to idempotently manage a key we effectively use the description field as a name field.

usage
Currently this field can only be set to `ENCRYPT_DECRYPT` (which is the default).

policy
An IAM policy describing which users can access this key.

class Alias
An alias for referring to a KMS key.

name
A name to refer to this alias by.

key
A *Key* to point this alias at.

class Grant
Grant access to a KMS key by AWS principal.

name

grantee_principal

retiring_principal

operations
Must be one or more of:

- `Decrypt`
- `Encrypt`
- `GenerateDataKey`

- `GenerateDataKeyWithoutPlaintext`
- `ReEncryptFrom`
- `ReEncryptTo`
- `CreateGrant`
- `RetireGrant`

`encryption_context`

`encryption_context_subset`

`grant_tokens`

Lambda

Automatic build and deploy of python lambda zips

In order to avoid updating lambda frequently we have 2 goals for any system that produces zips to upload:

- Reproducible builds are important. If the Sha256 hash does not change then we don't have to upload. This is fairly straightforward with Python, unless binary `.so` files are involved.
- We don't want to run the build process if nothing has changed. A build system like `make` can use simple timestamps to tell if your build target is older than your build sources and automatically build that parts that have changed.

We assume that you have a project with a `setup.py` and `requirements.txt`. Let's write a `Makefile`. First of all we define some directories for the build to happen in:

```
src_dir=$(shell pwd)
build_dir=$(src_dir)/build
wheel_dir=$(src_dir)/wheelhouse
output_wheel_dir=$(build_dir)/wheels-to-deploy
output_tree_dir=$(build_dir)/output-tree
output_zip=$(build_dir)/lambda.zip
wheelhouse_stamp=$(build_dir)/wheelhouse-stamp
staging_stamp=$(build_dir)/staging-stamp
staging_tree_stamp=$(build_dir)/staging-tree-stamp
build_date=$(shell git log --date=local -1 --format="@%ct")
```

All good make files have an `all` that defines which targets to build if you just run `make`. And they declare a `.PHONY` target. They are targets that aren't on the file system and should always be evaluated. If `clean` wasn't a `.PHONY` then a file called `clean` might confuse `make` - it would think it was responsible for building the file called `clean`!:

```
all: $(output_zip)
.PHONY: all clean
```

Our `wheelhouse-stamp` target will build a `pip` wheelhouse of all our requirements. By building wheels we precompile them. Wheels are zips that we can just extract and combine into a lambda zip. By creating a stamp file `make` can determine if the wheelhouse is older than the `requirements.txt`:

```
$(wheelhouse_stamp): $(src_dir)/requirements.txt
    @echo "Building wheels into wheelhouse..."
    pip wheel -q -r requirements.txt . --wheel-dir=$(wheel_dir) --find-links=$(wheel_
↪dir)
    touch $@
```

With the current state of tooling it is quite hard to build wheels twice and get byte identical output. So as a workaround right now you can keep this wheelhouse between builds. But then if the versions change or a dependency is removed our wheelhouse has stuff we don't want. So we have a temporary intermediate wheelhouse. Every time we update it we delete it first. It reuses the wheels from the caching wheelhouse so is fast and allows for idempotency:

```
$(staging_stamp): $(src_dir)/requirements.txt $(wheelhouse_stamp)
    @echo "Collecting wheels that match requirements..."
    rm -rf $(output_wheel_dir)
    pip wheel -q -r requirements.txt . --wheel-dir=$(output_wheel_dir) --find-links=
↪$(wheel_dir)
    touch $@
```

Now we need to unpack all the wheels we have collected. This is also where you would customize the output to add in extra files. We pin the max time stamp. This is because any directories that are created will have `$NOW` as their timestamp and this will wreck idempotence:

```
$(staging_tree_stamp): $(staging_stamp)
    rm -rf $(output_tree_dir)
    unzip -q "$(output_wheel_dir)/*.whl" -d $(output_tree_dir)
    find "$(output_tree_dir)" -newermt "$(build_date)" -print0 | xargs -0r touch --no-
↪dereference --date="$(build_date)"
    touch $@
```

Finally zip everything up. `-X` is crucial for idempotency and avoids setting various bits of extended metadata in the zip that are not reproducible and are unused:

```
$(output_zip): settings.json $(staging_tree_stamp)
    rm -f $(output_zip)
    cd $(output_tree_dir) && zip -q -X -9 -r $(output_zip) *
```

We need a clean rule as well to remove the stamp files and the other build artifacts:

```
clean:
    rm -f $(staging_tree_stamp) $(staging_stamp) $(wheelhouse_stamp)
    rm -f $(output_zip)
    rm -rf $(output_tree_dir)
```

Running `make` will now generate your `lambda.zip` ready to upload. Running `make` again should be a no-op. This means we can use `make -q` to create an idempotent lambda bundle. So in your `Touchdownfile`:

```
bundle = self.workspace.add_fuselage_bundle(
    target=self.workspace.add_local()
)

bundle.add_execute(
    command="make",
    unless="make -q",
)

self.aws.add_lambda_function(
    name="myfunction",
    role=self.aws.get_role(name="myrole"),
    handler="mymodule.myfunction",
    code=bundle.add_output(name="lambda.zip"),
)
```

How would do I rebuild the zip when my local source changes?

If your project has a folder called `myproject` full of `.py` files then you can use `find` to build a list of dependencies and then use those dependencies to trigger a rebuild of the wheels:

```
project_files = $(shell find $(src_dir)/myproject/ -type f -name '*.py')

$(wheelhouse_stamp): $(src_dir)/requirements.txt $(project_files)
    @echo "Building wheels into wheelhouse..."
    pip wheel -q -r requirements.txt . --wheel-dir=$(wheel_dir) --find-links=$(wheel_
↪dir)
    touch $@
```

If you don't want to use `pip` for your project, only your requirements, you can use `cp` and copy your `myproject` folder in instead:

```
project_files = $(shell find $(src_dir)/myproject/ -type f -name '*.py')

$(staging_tree_stamp): $(staging_stamp) $(project_files)
    rm -rf $(output_tree_dir)
    unzip -q "$(output_wheel_dir)/*.whl" -d $(output_tree_dir)
    cp -a $(src_dir)/myproject $(output_tree_dir)/myproject
    find "$(output_tree_dir)" -newermt "$(build_date)" -print0 | xargs -0r touch --no-
↪dereference --date="$(build_date)"
    touch $@
```

How can I copy settings into my lambda.zip?

You can use the fuselage file resource to generate a json file. Give an SQS queue called `myqueue`:

```
bundle.add_file(
    name="settings.json",
    contents=serializers.Json(serializers.Dict(
        AWS_SQS_URL=myqueue.get_property("QueueUrl"),
    )),
)

bundle.add_execute(
    command=os.path.join(os.getcwd(), "make"),
    unless=os.path.join(os.getcwd(), "make -q"),
    env={
        "PATH": os.path.join(sys.prefix, "bin") + ":/usr/local/sbin:/usr/local/bin:/
↪usr/sbin:/usr/bin:/sbin:/bin",
    },
    cwd=os.getcwd(),
)
```

This will ensure that the queue is created before generating the `settings.json` that refers to it, and then create a `settings.json` which can be picked up by `make`:

```
$(staging_tree_stamp): $(staging_stamp) settings.json
    rm -rf $(output_tree_dir)
    unzip -q "$(output_wheel_dir)/*.whl" -d $(output_tree_dir)
    cp $(src_dir)/settings.json $(output_tree_dir)/settings.json
    find "$(output_tree_dir)" -newermt "$(build_date)" -print0 | xargs -0r touch --no-
↪dereference --date="$(build_date)"
```

```
    touch $@
```

Your lambda function can then do something like this:

```
import os
FUNCTION_DIRECTORY = os.path.dirname(__file__)
globals().update(json.loads(open(os.path.join(FUNCTION_DIRECTORY, "settings.json"))))
```

And all the keys in the settings files will now be available like any other global variable.

Function

class Function

You can register a lambda function against an Amazon account resource:

```
def hello_world(event, context):
    print event

aws.add_lambda_function(
    name = 'myfunction',
    role = aws.add_role(
        name='myrole',
        #..... snip ....,
    ),
    code=hello_world,
    handler="main.hello_world",
)
```

name

The name for the function, up to 64 characters.

description

A description for the function. This is shown in the AWS console and API but is not used by lambda itself.

role

A *Role* resource.

The IAM role that Lambda assumes when it executes your function to access any other Amazon Web Services (AWS) resources.

code

A Zip file as bytes.

This can be a python callable. For example:

```
def hello_world(event, context):
    print event

aws.add_lambda_function(
    name='hello_world',
    code=hello_world,
    handler='main.hello_world'
    ...
)
```

It must take 2 arguments only - `event` and `context`.

This is intended for proof of concept demos when first starting out with lambda - there is no mechanism to ship dependencies of this function, it is literally the output of *inspect.getsource()* that is uploaded.

s3_file
An S3 `File`.

A new version of the lambda function is published when touchdown detects that the date/time stamp of this file is newer than the last modified stamp on the lambda function.

handler
The entry point to call.

For the `python2.7` runtime with a `shrink_image.py` module containing a function called `handler` the handler would be `shrink_image.handler`.

For the `node` runtime with a `CreateThumbnail.js` module containing an exported function called `handler`, the handler is `CreateThumbnail.handler`.

For the `java8` runtime, this would be something like `package.class-name.handler` or just `package.class-name`.

timeout
An integer. The number of seconds (between 1 and 300) that a lambda function is allowed to execute for before it is interrupted. The default is 3 seconds.

memory
The amount of RAM your lambda function is given. The amount of CPU is assigned based on this as well - more RAM means more CPU is allocated.

The default value is 128mb, which is also the minimum. Can assign up to 1536mb.

publish

Relational Database Service

Database

class Database

name
The name of the database server instance. This must be unique within your account/region and is required.

db_name
The name of a database to create in this instances.

allocated_storage
The amount of storage to be allocated (in GB). This must be 5 or more, and less than 3072. The default is 5.

iops

instance_class
The kind of hardware to use, for example `db.t1.micro`

engine
The type of database to use, for example `postgres`

engine_version

license_model

master_username
The username of the main client user

master_password
The password of the main client user

security_groups
A list of security groups to apply to this instance

publically_accessible

availability_zone

subnet_group
A `SubnetGroup` resource.

preferred_maintenance_window

multi_az

storage_type

storage_encrypted
Specifies whether or not he database instance has encrypted storage. By default it does not.

If set to true you can also set `key` to a KMS key. If you do not the default KMS key is used.

allow_major_version_upgrade

auto_minor_version_upgrade

character_set_name

backup_retention_period

preferred_backup_window

license_model

port

paramter_group
A `ParameterGroup` resource. Not currently supported.

option_group
A `OptionGroup` resource. Not currently supported.

apply_immediately

SubnetGroup

class Database

name

description

subnets
A list of `touchdown.vpc.Subnet` resources that database nodes can exist in.

Route53

HostedZone

class HostedZone

You can add a Route53 hosted zone from an AWS account resource:

```
aws.add_hosted_zone(
    name="example.com",
    records=[
        {"name": "www", "type": "A", "alias": my_load_balancer},
    ],
)
```

name
The name of the hosted zone.

comment
A comment about the hosted zone that is shown in the AWS user interface.

records
A list of *Record* resources.

shared
Set this to `True` in the zone is not exclusively managed by this touchdown configuration. Otherwise shared zones may be unexpectedly deleted.

vpc
Set this to a `Vpc` in order to create a private hosted zone.

DNS records

class Record

name
For example, `www`. This field is required.

type
The type of DNS record. For example, `A` or `CNAME`. This field is required.

set_identifier
When using weighted recordsets this field differentiates between records for `name`/`type` pairs. It is only required in that case.

ttl
How long the DNS record is cacheable for, in seconds.

values
A list of values to return when a client resolves the given `name` and `type`.

alias
If creating an `A` record you can pass in one of the follwing to create an alias record. This acts like a server side CNAME. Route53 resolves the domain name and returns IP addresses directly, reducing latency.

You can pass in:

- A *LoadBalancer* instance
- A CloudFront *Distribution*
- A CloudFront *StreamingDistribution*

Simple storage service

class Bucket

A bucket in the Amazon S3 service.

Can be added to any account resource:

```
bucket = aws.add_bucket(
    name='my-bucket',
)
```

name

The name of the bucket. This field is required, and it must be unique for the whole of Amazon AWS.

region

The region of the bucket. The default is to create the bucket in the sane region as the region specified by the account.

accelerate

Set this to `Enabled` to enable Transfer Acceleration.

rules

A list of CORS rules:

```
aws.add_bucket(
    name="my-test-bucket",
    rules=[{
        "allowed_methods": ["PUT", "POST", "GET"],
        "allowed_origins": ["*"],
        "allowed_headers": ["content-md5"],
        "expose_headers" :["ETag"],
        "max_age_seconds": 3000,
    }],
)
```

policy

An S3 bucket policy string:

```
aws.add_bucket(
    name="my-test-bucket",
    policy= json.dumps({
        "Version":"2008-10-17",
        "Statement":[{
            "Sid":"AllowFullControlForBucketOwner",
            "Effect":"Allow",
            "Principal":{"AWS":"arn:aws:iam::111111111111:root"},
            "Action":"s3:*",
            "Resource":"arn:aws:s3:::my-test-bucket/*"}],
    }),
)
```

notify_lambda

A list of lambda functions to call when a notification event occurs. For example:

```
mybucket = aws.add_bucket(
    name='my-bucket',
    notify_lambda=[{
        "name": "process-new-objects",
```

```
        "function": myfunction,
        "events": ["s3:ObjectCreated:*"]
    }]
)
```

Adding files to buckets

class File

Touchdown has basic support for pushing files to S3. This is lightweight and basic. It's using for setting up things like crossdomain.xml:

```
bucket = aws.add_bucket(name="my-test-bucket")
bucket.add_file(
    name="crossdomain.xml",
    contents=open("crossdomain.xml").read(),
    acl="public-read",
)
```

name

contents

Simple Notification Service

Simple Notification Service is a managed push notification service.

It can push notifications to:

- HTTP endpoints
- Amazon SQS *Queue*'s
- Amazon Lambda *Function*'s
- SMS text messages
- E-mail
- Apple, Android, Fire OS and Window devices

Messages published to SNS are stored redundantly to prevent messages being lost.

Note: Accessing this service requires internet access.

If you want to access this from an EC2 you must either:

- Give the node a public IP and connect its route table to an internet gateway
- Set up NAT
- Set up a proxy cluster

---

class Topic

An SNS topic.

Can be added to any account resource:

```
topic = aws.add_topic(
    name='my-bucket',
)
```

name
The name of the bucket. This field is required, and it must be unique for the whole of Amazon AWS.

notify
A list of resources that should be subscribed to this topic. Can be any of:

- *Queue*
- *Function*

display_name

policy

delivery_policy

Simple Queue Service

Simple Queue Service is a managed queue service.

It is considered to be engineered for redundancy, so you do not need to set up extra queues for availabilty.

Note: Accessing this service requires internet access.

If you want to access this from an EC2 you must either:

- Give the node a public IP and connect its route table to an internet gateway
- Set up NAT
- Set up a proxy cluster

---

class Queue
An SQS Queue.

Can be added to any account resource:

```
queue = aws.add_queue(
    name='my-queue',
)
```

name
The name of the queue.

delay_seconds
An integer between 0 and 900.

maximum_message_size
An integer between 1024 and 252144.

message_retention_period
An integer between 60 and 1209600

policy

receive_message_wait_time_seconds
An integer between 0 and 20.

visibility_timeout
An integer between 0 and 43200. The default is 30.

Virtual private clouds

Virtual Private Clouds

class VPC
A Virtual Private Cloud in an Amazon region.

VPC's let you logically isolate components of your system. A properly defined VPC allows you to run most of your backend components on private IP addresses - shielding it from the public internet.

You define the IP's available in your VPC with a CIDR-form IP address.

You can add a VPC to your workspace from any Amazon account resource:

```
account = workspace.add_aws(
    access_key_id='....',
    secret_access_key='....',
    region='eu-west-1',
)

vpc = workspace.add_vpc(
    name='my-first-vpc',
    cidr_block='10.0.0.0/16',
)
```

name
The name of the VPC. This field is required.

cidr_block
A network range in CIDR form. For example, 10.0.0.0/16. A VPC network should only use private IPs, and not public addresses. This field is required.

tenancy
This controls whether or not to enforce use of single-tenant hardware for this VPC. If set to `default` then instances can be launched with any tenancy options. If set to `dedicated` then all instances started in this VPC will be launched as dedicated tenancy, regardless of the tenancy they requsest.

tags
A dictionary of tags to associate with this VPC. A common use of tags is to group components by environment (e.g. "dev1", "staging", etc) or to map components to cost centres for billing purposes.

If you create a dedicated VPC for your application instead of using the default VPC then you must create at least one `Subnet` in it.

Subnets

class Subnet
Subnets let you logically split application reponsibilities across different network zones with different routing rules and ACL's. You can also associate a subnet with an availability zone when building H/A solutions.

You can add a subnet to any VPC:

```
subnet = vpc.add_subnet(
    name='my-first-subnet',
```

```
    cidr_block='10.0.0.0/24',
)
```

name
The name of the subnet. This field is required.

cidr_block
A network range specified in CIDR form. This field is required and must be a subset of the network range covered by the VPC. For example, it cannot be 192.168.0.0/24 if the parent VPC covers 10.0.0.0/24.

network_acl
A *NetworkACL* resource.

This controls which IP address a subnet can connect out to an can receive connections from.

route_table
A *RouteTable* resource.

Where to route traffic external to the VPC. This controls whether to send traffic via an internet gateway, vpn gateway or via another instance that is applying NAT to traffic.

availability_zone
The AWS availability zone this subnet is created in.

tags
A dictionary of tags to associate with this VPC. A common use of tags is to group components by environment (e.g. "dev1", "staging", etc) or to map components to cost centres for billing purposes.

In order for a subnet to access the internet it will need a `RouteTable` attaching to it with an `InternetGateway`.

Security Groups

class SecurityGroup
Resources can be placed in SecurityGroup resources. A SecurityGroup then applies a set of rules about what incoming and outgoing traffic is allowed.

You can create a SecurityGroup in any VPC:

```
security_group = vpc.add_security_group(
    name='my-security-group',
    ingress=[dict(
        protocol='tcp',
        from_port=22,
        to_port=22,
        network='0.0.0.0/0',
    )],
)
```

name
The name of the security group. This field is required.

description
A short description of the SecurityGroup. This is shown in the AWS console UI.

ingress
A list of *Rule* resources describing what IP's or components are allowed to access members of the security group.

egress
A list of :class: *Rule* resources describing what IP's or components can be access by members of this security group.

tags
A dictionary of tags to associate with this VPC. A common use of tags is to group components by environment (e.g. "dev1", "staging", etc) or to map components to cost centres for billing purposes.

Defining rules

class Rule
Represents a rule in a security group.

You shouldn't create `Rule` resources directly, they are created implicitly when defining a *SecurityGroup*. For example:

```
security_group = vpc.add_security_group(
    name='my-security-group',
    ingress=[
        {"port": 80, "network": "0.0.0.0/0"},
        {"port": 443, "network": "0.0.0.0/0"},
    ],
)
```

This will implicitly create 2 `Rule` resources.

protocol
The network protocol to allow. It must be one of `tcp`, `udp` or `icmp`. It is `tcp` by default.

port
The port to allow access to. You might want to specify a range instead. In that case you can set `from_port` and `to_port` instead.

security_group
The :class: `SecurityGroup` that this rule is about. You cannot specify `security_group` and `network` on the same rule.

Network ACL's

class NetworkACL
Network ACL's provide network filtering at subnet level, controlling both inbound and outbound traffic. They are:

- Stateless. This means that return traffic is not automatically allowed. This can make them more difficult to set up.
- Attached to the subnet. So you don't have to specify them when starting an instance.
- Processed in the order specified. The first match is the rule that applies.
- Supports ALLOW and DENY rules.

Any traffic that doesn't match any rule is blocked.

You can create a NetworkACL in any VPC:

```
network_acl = vpc.add_network_acl(
    name='my-network-acl',
    inbound=[dict(
```

```
        protocol='tcp',
        port=22,
        network='0.0.0.0/0',
    )],
)
```

Network ACL's are updated by replacement. This means each time a change is detected an entirely new one will be created and subnets using the old one (that are managed by touchdown) will be pointed at the new one. This avoids having to re-number rules in an existing ACL and makes rolling back easier.

name
The name of the network acl. This field is required.

tags
A dictionary of tags to associate with this VPC. A common use of tags is to group components by environment (e.g. "dev1", "staging", etc) or to map components to cost centres for billing purposes.

Defining rules

class Rule
Represents a rule in a *NetworkACL*.

You shouldn't create `Rule` resources directly, they are created implicitly when defining a *NetworkACL*. For example:

```
network_acl = vpc.add_network_acl(
    name='my-network-acl',
    inbound=[
        {"port": 80, "network": "0.0.0.0/0"},
        {"port": 443, "network": "0.0.0.0/0"},
    ],
)
```

This will implicitly create 2 `Rule` resources.

There is always a default catch-all rule that denies any traffic you haven't added a rule for.

Route Tables

class RouteTable
A route table contains a list of routes. These are rules that are used to determine where to direct network traffic.

A route table entry consists of a destination cidr and a component to use when to route that traffic. It is represented in touchdown by a *Route* resource.

You can create a route table in any vpc:

```
vpc.add_route_table(
    name="internet_access",
    subnets=[subnet],
    routes=[dict(
        destination_cidr='0.0.0.0/0',
        internet_gateway=internet_gateway,
    )]
)
```

name
The name of the route table. This field is required.

routes
A list of *Route* resources to ensure exist in the route table.

propagating_vpn_gateways
A list of `VpnGateway` resources that should propagate their routes into this route table.

tags
A dictionary of tags to associate with this VPC. A common use of tags is to group components by environment (e.g. "dev1", "staging", etc) or to map components to cost centres for billing purposes.

Defining routes

class Route
Represents a route in a route table.

You shouldn't create Route resources directly, they are created implicitly when defining a *RouteTable*. For example:

```
vpc.add_route_table(
    name="internet_access",
    routes=[
        {"destionation_cidr": "8.8.8.8/32", "internet_gateway": ig},
        {"destionation_cidr": "8.8.4.4/32", "internet_gateway": ig},
    ]
)
```

You should specify 2 attributes: `destination_cidr` and where to route that traffic.

destination_cidr
A network range that this rule applies to in CIDR form. You can specificy a single IP address with `/32`. For example, `8.8.8.8/32`. To apply a default catch all rule you can specify `0.0.0.0/0`. ""“

internet_gateway
A `InternetGateway` resource.

nat_gateway
A `NatGateway` resource.

Internet Gateway

class InternetGateway
An internet gateway is the AWS component that allows you to physically connect your VPC to the internet. Without an internet gateawy connected to your VPC then traffic will not reach it, even if assigned public IP addresses.

You can create an internet gateway in any VPC:

```
internet_gateway = vpc.add_internet_gateway(
    name='my-internet-gateway',
)
```

name
The name of the gateway. This field is required.

tags
A dictionary of tags to associate with this VPC. A common use of tags is to group components by environment (e.g. "dev1", "staging", etc) or to map components to cost centres for billing purposes.

NAT Gateway

class NatGateway
An internet gateway is the AWS component that allows you connect a private VPC to the internet.

You can create a NAT gateway in any subnet:

```
nat_gateway = subnet.add_nat_gateway(
    elastic_ip=....,
)
```

name
You cannot assign a name to a NAT Gateway - it automatically inherits the name of the subnet it is placed in (i.e. its *Name* tag).

elastic_ip

Hardware VPN

Amazon provide a hardware VPN facility for connecting your VPC to your corporate datacenter over industry standard ipsec encryption. This is a dial-in service. You connect to it, it does not connect to you.

VPN Connections

class VpnConnection
You can create a VPN Connection in any VPC:

```
vpn = vpn.add_vpn_connection(
    name='my-vpn-connection',
)
```

By default you can only create 10 VPN connections within an Amazon account.

name
The name of the vpn connection. This field is required.

customer_gateway
A *CustomerGateway*. This field is required.

vpn_gateway
A *VpnGateway*. This field is required.

type
The type of `VpnConnection` to create. The default is `ipsec.1`. This is also the only currently supported value.

static_routes_only
Set to True to only consider the routes defined in `static_routes`.

static_routes
A list of ip ranges in CIDR form.

tags
A dictionary of tags to associate with this VPC. A common use of tags is to group components by environment (e.g. "dev1", "staging", etc) or to map components to cost centres for billing purposes.

Customer Gateway

class CustomerGateway
A CustomerGateway represents the non-Amazon end of a VpnConnection.

You can create an customer gateway in any VPC:

```
customer_gateway = vpc.add_customer_gateway(
    name='my-customer-gateway',
)
```

name
The name of the customer gateway. This field is required.

type
The type of `CustomerGateway` to create. The default is `ipsec.1`. This is also the only currently supported value.

public_ip
The internet-routable IP address for the customer gateway's outside interface.

bgp_asn
For devices that support BGP, the gateway's BGP ASN.

tags
A dictionary of tags to associate with this VPC. A common use of tags is to group components by environment (e.g. "dev1", "staging", etc) or to map components to cost centres for billing purposes.

VPN Gateway

class VpnGateway
A VpnGateway represents the Amazon end of a VpnConnection.

You can create an vpn gateway in any VPC:

```
vpn_gateway = vpc.add_vpn_gateway(
    name='my-vpn-gateway',
)
```

name
The name of the vpn gateway. This field is required.

type
The type of `CustomerGateway` to create. The default is `ipsec.1`. This is also the only currently supported value.

availability_zone
The availability zone to place the Vpn Gateway in.

tags
A dictionary of tags to associate with this VPC. A common use of tags is to group components by environment (e.g. "dev1", "staging", etc) or to map components to cost centres for billing purposes.

Web Application Firewall

Amazon provide a Web Application Firewall for your CloudFront Web Distributions. At 'layer 7' it is able to inspect HTTP traffic passing through CloudFront and block malicious signatures or even just provide IP filtering that is URI specific.

How do I create an IP whitelist for my staging environment?

Let's say your office IP address is 8.8.8.8. You need to create an *ip_set* with the addresses that are allowed to access your staging environment:

```
ip_set = self.aws.add_ip_set(
    name="site-access-permitted",
    addresses=[
        "8.8.8.8/32",
    ],
)
```

We add a *rule* that says matches all addresses in that set. With no other `predicates` defined this will match all HTTP traffic from the addresses in the set:

```
authorized_access = self.aws.add_rule(
    name="authorized-access",
    predicates=[{"ip_set": ip_set}],
    metric_name="AuthorizedAccess",
)
```

The final step is to add this to a web_acl and tell WAF that the rule should `ALLOW` traffic matching it, and all other traffic should be blocked:

```
staging_firewall = self.aws.add_web_acl(
    name="staging-firewall",
    activated_rules=[{
        "rule": authorized_access,
        "priority": 1,
        "action": "ALLOW",
    }],
    default_action="BLOCK",
    metric_name="MyWafRules",
)
```

If you are using Touchdown to manage your CloudFront distribution you can use the `web_acl` attribute to link it all up:

```
self.aws.add_distribution(
    name="www.example.com",
    web_acl=my_web_acl,
)
```

How do I IP restrict my admin interface?

Let's say your admin interface is located at */admin* and your office IP address is 8.8.8.8. You need to create a byte_match to match requests for the URI and an ip_set to match your office IP:

```
byte_match_set = self.aws.add_byte_match_set(
    name="dashboard-access",
    byte_matches=[{
        "field": "URI",
        "transformation": "URL_DECODE",
        "position": "STARTS_WITH",
        "target": "/admin/",
    }],
)

ip_set = self.aws.add_ip_set(
    name="dashboard-access-permitted",
    addresses=[
        "8.8.8.8/32",
    ],
)
```

And we want to match requests that *aren't* from our *ip_set* but do match our *byte_match_set*:

```
unauthorised_admin_access = self.aws.add_rule(
    name="unauthorized-admin-access",
    predicates=[
        {
            "ip_set": ip_set,
            "negated": True,
        },
        {
            "byte_match_set": byte_match_set,
        }
    ],
    metric_name="UnauthorizedAdminAccess",
)
```

The final step is to add this to a web_acl and tell WAF that the rule should `BLOCK` traffic matching it:

```
my_web_acl = self.aws.add_web_acl(
    name="my-waf-rules",
    activated_rules=[{
        "rule": unauthorised_admin_access,
        "priority": 1,
        "action": "BLOCK",
    }],
    default_action="ALLOW",
    metric_name="MyWafRules",
)
```

Web ACL

class `WebACL`

To create a Web ACL you need to specify at least its `name`, `metric_name` and `default_action`:

```
web_acl = aws.add_web_acl(
    name='my-webacl',
    metric_name='MyWebACL',
    default_action='BLOCK',
)
```

name
The name of the Web ACL. This field is required.

activated_rules
A list of rules that apply to this ACL. The following 3 fields must be set:

rule A :py:class:~'Rule'.

priority Rules with lower `priority` are evaluated before rules with a higher `priority`.

action Must be one of `ALLOW`, `BLOCK` or `COUNT`.

default_action
The default action to take if no rules in `activated_rules` have matched the request. Must be one of `ALLOW` or `BLOCK`.

metric_name
A CloudWatch metric name.

Rule

class Rule
To create a WAF Rule you need to specify its `name` and a `metric_name`:

```
rule = aws.add_rule(
    name='my-waf-rule',
    metric_name='MyWafRule',
)
```

name

metric_name

IP Set

class IPSet
To get started with IP sets you at least need to give it a `name`:

```
ips = aws.add_ip_set(
    name='my-ips',
)
```

name
The name of the ip_set. This must be unique within a region.

addresses
A list of IP networks to match against:

```
ips = aws.add_ip_set(
    name='my-ips',
    addresses=[
        '8.8.8.8/32',
    ]
)
```

As a CloudFront distribution can only be accessed from the public internet these should be public addresses. IP's in the following networks are not valid:

- `10.0.0.0/8`

- 172.16.0.0/12
- 192.168.0.0/16

Byte Match Set

class ByteMatchSet

To create a byte match set you need to at least gitve it a `name`:

```
byte_matches = aws.add_byte_match_set(
    name='my-byte-matches',
)
```

name

The name of the byte_match_set. This must be unique within a region.

byte_matches

A list of data to match against:

byte_matches = aws.add_byte_match_set(name='my-byte-matches', byte_matches=[{

"field": "URI", "transformation": "URL_DECODE", "position": "STARTS_WITH", "target": "/admin/",

}],

)

field

Must be one of:

`URI` `QUERY_STRING` `HEADER` `METHOD`

Use this to limit your matches to a `GET` or `POST` method, etc.

BODY Match against the first 8192 bytes of he body of the request.

header

You can only use this attribute if `field` is set to `HEADER`.

transformation

A transformation to apply before comparing the selected `field` to `target`.

Must be one of:

`CMD_LINE` `COMPRESS_WHITE_SPACE` `HTML_ENTITY_DECODE` `LOWERCASE` `URL_DECODE` `NONE`

Don't apply any transformations to the string before matching against it.

The default value is `NONE`.

position

Where in the chosen `field` to look for `target`. Must be one of:

`CONTAINS` `CONTAINS_WORD` `EXACTLY` `STARTS_WITH` `ENDS_WITH`

target

Some byte data to look for in the chosen `field` after applying a `transformation`. Must be between 1 and 50 bytes.

Provisioner

Script

The provisioner can deploy a script to a target and execute it. This is great for simple deployments.

class `Script`

You can provision with a script from the workspace:

```
script = workspace.add_script(
    script=(
        "#! /bin/bash\n"
        "echo 'hello'\n"
    ),
    target={
        "hostname": "localhost",
        "username": "user",
    }
)
```

`script`

A script to copy to the host and run. This could be any thing the target knows how to execute. For example:

```
workspace.add_script(
    script=(
        "#! /usr/bin/env python\n"
        "print('hello from python')\n"
    ),
)
```

`target`

The target of the deployment. For example:

```
script = workspace.add_script(
    target={
        "hostname": "localhost",
        "username": "user",
    }
)
```

See `Provisioner` for more examples.

Fuselage

You can use Fuselage to deploy configuration changes to servers created and managed by Touchdown. Fuselage provides a pythonic API for building bundles of configuration which can be deployed idempotently on any system with a python interpreter.

class `Bundle`

You can create a bundle from the workspace:

```
bundle = workspace.add_fuselage_bundle(
    target={
        "hostname": "localhost",
        "username": "user",
    }
```

```
)

bundle.add_file(
    name="/etc/apt/sources.list",
    contents="...",
)
```

target
The target of the deployment. You can create a bundle from the workspace:

```
bundle = workspace.add_fuselage_bundle(
    target={
        "hostname": "localhost",
        "username": "user",
    }
)
```

See `Provisioner` for more examples.

Resources

Once you have a bundle you can add fuselage resources to it.

The following resources are available:

- File
- Line
- Directory
- Link
- Execute
- Checkout
- Package
- User
- Group
- Service

The fuselage documentation gives examples in the form:

```
Line(
    name="/etc/selinux/config",
    match=r"^SELINUX",
    replace="SELINUX=disabled",
)
```

You can do this in touchdown as follows:

```
bundle.add_line(
    name="/etc/selinux/config",
    match=r"^SELINUX",
    replace="SELINUX=disabled",
)
```

You might deploy minecraft to a server like this:

```
bundle = workspace.add_fuselage_bundle(
    target={
        "hostname": "example.com",
        "username": "deploy",
    }
)
bundle.add_directory(
    name='/var/local/minecraft',
)
bundle.add_execute(
    command='wget https://s3.amazonaws.com/Minecraft.Download/versions/1.8/minecraft_
↪server.1.8.jar',
    cwd="/var/local/minecraft",
    creates="/var/local/minecraft/minecraft_server.1.8.jar",
)
bundle.add_file(
    name='/var/local/minecraft/server.properties',
    contents=open('var_local_minecraft_server.properties').read(),
)
bundle.add_file(
    name="/etc/systemd/system/minecraft.service",
    contents=open("etc_systemd_system_minecraft.service"),
)
bundle.add_execute(
    command="systemctl daemon-reload",
    watches=['/etc/systemd/system/minecraft.service'],
)
bundle.add_execute(
    command="systemctl restart minecraft.service",
    watches=[
        "/var/local/minecraft/server.properties",
        "/etc/systemd/system/minecraft.service",
    ]
)
```

Targets

The provisioner can target multiple systems. It's primary mechanisms for provisioning are:

- Localhost direct access
- SSH (including jump-off hosts)

class `Provisioner`

You cannot directly add a provisioner to your workspace. You must add a specific type of provisioner to your workspace:

- `Fuselage`
- `Bash`

However all provisioner types support the attributes below.

`target`

The target of the deployment.

You can target your local machine directly. This won't use SSH. It's a dedicated transport that runs locally:

```
bundle = workspace.add_fuselage_bundle(
    target=workspace.add_local(),
)
```

You can provide SSH connection details:

```
bundle = workspace.add_fuselage_bundle(
    target={
        "hostname": "localhost",
        "username": "user",
    }
)
```

This will SSH to `localhost` as user `user` to execute the bundle. You can chain connections (a technique called jump-off hosts) to traverse bastions:

```
bundle = workspace.add_fuselage_bundle(
    target={
        "hostname": "host1",
        "username": "user",
        "proxy": {
            "hostname": "host2",
            "username": "fred",
        }
    },
)
```

When used like this a connection will be made to host2. From there a second connection will be made from host2 to host1. This will be tunneled inside the first connection using the `direct-tcpip` feature of SSH.

Instead of passing a `hostname` you can pass `instance`. This lets you connect to resources defined elsewhere in your configuration. This even works on *`AutoScalingGroup`* instances!:

```
application_servers = aws.add_auto_scaling_group(
    name='my-application-servers',
)

bundle = workspace.add_fuselage_bundle(
    target={
        "instance": application_servers,
        "username": "user",
    },
)
```

Notifications

New Relic deployment notifications

class NewRelicDeploymentNotification

You can send a notification from the workspace:

```
newrelic = workspace.add_newrelic_deployment_notification(
    apikey="XXXXXXXXXXXXXXXXXXXXXXXX",
    app="myapp-staging",
```

```
    revision="3.1.0"
)
```

This uses the New Relic Deployment Notification REST API.

apikey
Your NewRelic API key. This is seperate from your licence key. Required.

app
The name of the application to record the deployment against. Required.

revision
The version of software that was just deployed. Required. Max 127 characters.

description
A description of the change. Max 65535 characters.

changelog
A copy of the changelog to attach to this deployment record. Max 65535 characters.

user
The user that pushed this changed. Max 31 characters.

Slack notifications

To get a slack notification you'll need to add the "Incoming Webhook" to your account. You'll be given a URL that looks like this:

```
https://hooks.slack.com/services/T00000000/B00000000/XXXXXXXXXXXXXXXXXXXXXXXX
```

Treat this URL as one of your application secrets.

class SlackNotification
You can send a notification from the workspace:

```
script = workspace.add_slack_notification(
    webhook="https://hooks.slack.com/services/T00000000/B00000000/
↪XXXXXXXXXXXXXXXXXXXXXXXX",
    channel="#touchdown",
    text="Hello from touchdown",
)
```

webhook
A `hooks.slack.com` url to post notifications to.

username
The username of the bot that posts this in the channel. Messages will appear to come from this user, even if there isn't a user with this name. The default user is `yaybu`.

icon_url
A url to fetch an avatar from for this bot user.

icon_emoji
A slack emoji to use as an avatar, for example `:ghost:`. Should not be used at the same time as `icon_url`.

channel
The channel to post in. By default the integration will post in the channel defined by the hook itself. You can set it to a `#channel` or `@user` that you want to send notifications to.

text
A message to send to the channel. You can use a touchdown serializer to set this based on other resources you have defined.

attachments
A list of *Attachment*. These allow construction of prettier and more informative notifications.

Advanced notifications

class Attachment

fallback
The fallback message to show if the advanced notification is not shown (for example in mobile notifications or on IRC). This is required.

color
An optional value that can either be one of `good`, `warning`, `danger`, or any hex color code (eg. `#439FE0`). This value is used to color the border along the left side of the message attachment.

pretext
Optional text that appears above the message attachment block.

author_name
Small text used to display the author's name.

author_link
A valid URL that will hyperlink the author_name text mentioned above. Will only work if author_name is present.

author_icon
A valid URL that displays a small 16x16px image to the left of the author_name text. Will only work if author_name is present.

title
The title is displayed as larger, bold text near the top of a message attachment.

title_link
If set, the `title` text will appear hyperlinked.

text
This is the main text in a message attachment, and can contain standard message markup. The content will automatically collapse if it contains 700+ characters or 5+ linebreaks, and will display a "Show more..." link to expand the content.

fields
Metadata to show in a table inside the message attachment. Represented as a list of dictionaries:

```
workspace.add_slack_notification(
    #.. snip ..
    attachments=[{
        "fallback": "A deployment to production just completed",
        "fields": [{
            "title": "Environment",
            "value": "production",
            "short": True,
        }]
    }]
)
```

The fields are:

title Shown as a bold heading above the `value` text. It cannot contain markup and will be escaped for you.

value The text value of the field. It may contain standard message markup and must be escaped as normal. May be multi-line.

short An optional flag indicating whether the value is short enough to be displayed side-by-side with other values.

image_url
A valid URL to an image file that will be displayed inside a message attachment. Slack currently supports the following formats: GIF, JPEG, PNG, and BMP.

Large images will be resized to a maximum width of 400px or a maximum height of 500px, while still maintaining the original aspect ratio.

thumb_url
A valid URL to an image file that will be displayed as a thumbnail on the right side of a message attachment. Slack currently supports the following formats: GIF, JPEG, PNG, and BMP.

The thumbnail's longest dimension will be scaled down to 75px while maintaining the aspect ratio of the image. The filesize of the image must also be less than 500 KB.

markdown_in
Fields which have markdown in them that needs rendering. For example if `text` contains markdown you must do:

```
workspace.add_slack_notification(
    attachments=[{
        "text": "A deployment to ``production`` just completed",
        "markdown_in": ["text"],
    }]
)
```

Examples

For a post deployment notification that includes a changelog snippet you can do something like:

```
workspace.add_slack_notification(
    webhook="https://hooks.slack.com/services/T00000000/B00000000/
↪XXXXXXXXXXXXXXXXXXXXXXXX",
    channel="#touchdown",
    attachments=[{
        "fallback": "Deployment of '1.3' to 'production' completed",
        "title": "Deployment by user1 completed",
        "text": "\n".join([
            "```",
            "- Added a new foobar <user1>",
            "- Fixed the frobnicator <user2>"
            "```",
        ]),
        "markdown_in": ["text"],
        "fields": [{
            "title": "Environment",
            "value": "production",
            "short": True,
        }, {
```

```
            "title": "Version",
            "value": "1.3",
            "short": True,
        }],
    }],
)
```

Managing state and tunables

There are parts of your cloud configuration that are fixed parts of your design. Firewalls rules that isolate your subnets. That you have a database and some autoscaling clusters. But there are other settings that are more fluid, such as the size of the instances started by your autoscaling. These are your tunables.

There are some API's that don't allow Touchdown to store enough state to achieve idempotence. An example of this is an `ElasticIp`. You can allocate an IP for your deployment, but there is no way to tag or name it. In order to remember which IP was allocated for which purpose metadata needs to be stored out of band. This is state.

Touchdown has a mechanism for declaring these up front, defining validation and even allowing some tunables to be automatically generated the first time you deploy an environment. Using this you never need to manually generate SSH keys or a new django secret key for a new build again.

You can define a config file to store your state in from the workspace:

```
config = workspace.add_ini_file(
    file='foo/bar/baz.cfg',
)
```

This config will be stored in the folder `foo/bar` relative to your *Touchdownfile*.

You may have tunables such as passwords that you wish to store. For these you can add encryption. To use GPG encryption:

```
gpg = workspace.add_gpg(symmetric=True)
config = workspace.add_ini_file(
    file=gpg.add_cipher(file='foo/bar/baz.cfg')
)
```

The *gpg* object just represents a set of 'goals' for the GPG wrappers. In this case the goal is to prompt for a passphrase to use for symmetric encryption. Using *gpg.add_cipher* the local file is acquired, but it is filtered by the GPG engine. The config component can operate on the local file as normally, but GPG encryption and decryption is transparently applied as required.

Other systems can be plugged in as required. For example, you can use `Kms`. This uses the fernet encryption backend, with a secret key backed by HSM at Amazon.

You can use tunables to generate secret keys and pass them to an instance:

```
django_secret_key = config.add_string(
    name='django.secret_key',
    default=django_secret_key(),
    retain_default=True,
)

lc = aws.add_launch_configuration(
    user_data=serializers.Json({
        "DJANGO_SECRET_KEY": django_secret_key,
```

```
    }),
    ... snip ....,
)
```

You can use tunables to manage the capacity of an autoscaling group:

```
aws.add_autoscaling_group(
    name='web',
    min=config.add_integer(
        name='scaling.web.min',
        default=1,
    )
    max=config.add_integer(
        name='scaling.web.max',
        default=1,
    ),
    ... snip ...,
)
```

CHAPTER 6

Getting started

- **From the top:** *Overview* | *Installation*
- **CLI:** *The touchdown command* | *Applying changes* | *Tearing down environments* | *Viewing logs* | *Snapshotting your data* | *Rolling back data* | *SSHing to your infrastructure* | *SCPing to/from your infrastructure* | *Generating graphs*
- **Tutorial:** *Hello world* | *Django* | *Handling S3 Events with Lambda* | *A serverless redirect service*

CHAPTER 7

Resources

- **Amazon:** *Authentication* | *Autoscaling* | *Building serverless API's* | *CDN* | *Compute* | *DNS* | *Encryption Key Management* | *Key Value Stores* | *Identity & Access Management* | *Lambda zero-admin compute* | *Load Balancing* | *Monitoring* | *Networking* | *Relational Databases* | *Simple Notification Service* | *Simple Queue Service* | *Simple Storage Service* | *SSL Certificates* | *Transcoding* | *VPNs* | *Web Application Firewall*
- **Provisioning:** *Provisioner targets* | *Deploying scripts* | *Deploying fuselage bundles*
- **Notifications:** *Slack notifications* | *NewRelic deploy notifications*
- **Tunables:** *Configuration*

CHAPTER 8

Getting help

- Ask a question in the #yaybu IRC channel.
- Report a bug in our issue tracker.

CHAPTER 9

Contributing

Writing your first PR? Checkout out the style guide

If you are hacking on the AWS code check out our walkthrough of all the lovely helper classes here

Python Module Index

t

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

U

V

W