runway Documentation

Release 1.4.0

Onica Group

Feb 20, 2020

CONTENTS:

1 What is Runway? 1

2 Why use Runway? 3

2.1 Installation . . . 3
2.2 Getting Started Guide . . . 5
2.3 Quickstart Guides . . . 8
2.4 Commands . . . 14
2.5 Runway Config File . . . 21
2.6 Module Configurations . . . 35
2.7 CFNgin . . . 54
2.8 Lookups . . . 91
2.9 Defining Tests . . . 93
2.10 Repo Structure . . . 95
2.11 Terminology . . . 97
2.12 Developer Guide . . . 98
2.13 Apache License . . . 102

3 Indices and tables 107

Python Module Index 109

Index 111

CHAPTER

ONE

WHAT IS RUNWAY?

Runway is a lightweight wrapper around infrastructure deployment (e.g. CloudFormation, Terraform, Serverless) & linting (e.g. yamllint) tools to ease management of per-environment configs & deployment.

CHAPTER

TWO

WHY USE RUNWAY?

Very simple configuration to:

- Perform automatic linting/verification
- Ensure deployments are only performed when an environment config is present
- Define an IAM role to assume for each deployment
- Wrangle Terraform backend/workspace configs w/ per-environment tfvars
- Avoid long-term tool lock-in
  - Runway is a simple wrapper around standard tools. It simply helps to avoid convoluted Makefiles / CI jobs

2.1 Installation

To enable Runway to conform to our users' varying use cases, we have made it available via three different install methods - *cURL*, *npm*, and *pip*.

2.1.1 cURL

Arguably the easiest way to install Runway is by using curl. Use one of the endpoints below to download a single-binary executable version of Runway based on your operating system.

Operating System	Endpoint
Linux	https://oni.ca/runway/latest/linux
macOS	https://oni.ca/runway/latest/osx
Windows	https://oni.ca/runway/latest/windows

```
$ curl -L https://oni.ca/runway/latest/osx -o runway
```

Note: To install a specific version of Runway, you can replace `latest` with a version number.

Usage

To use the single-binary, run it directly as shown below. Please note that after download, you may need to adjust the permissions before it can be executed. (eg. macOS/Ubuntu:`chmod +x runway`)

```
$ ./runway deploy
```

Suggested use: CloudFormation or Terraform projects

2.1.2 npm

Runway is published on npm as `@onica/runway`. It currently contains binaries to support macOS, Ubuntu, and Windows.

While Runway can be installed globally like any other npm package, we strongly recommend using it per-project as a dev dependency. See *Why Version Lock Per-Project* for more info regarding this suggestion.

```
$ npm i -D @onica/runway
```

Usage

```
$ npx runway deploy
```

Suggested use: Serverless or AWS CDK projects

2.1.3 pip

Runway runs on Python 2.7 and Python 3.5+.

Runway is hosted on PyPI as the package named `runway`. It can be installed like any other Python package, but we instead strongly recommend using it per-project with pipenv. See *Why Version Lock Per-Project* for more info regarding this suggestion.

Suggested use: Python projects

Version Locking with Pipenv

In your project's directory, execute `pipenv install runway`. This will:

1. Update (creating if missing) a `Pipfile` file with your project's Runway dependency
2. Create a Python virtual environment (hidden in your user profile folder) dedicated to your project, with Runway installed in it
3. Update (creating if missing) a `Pipfile.lock` file containing the exact versions/crypto-hashes of Runway (and dependencies) installed in your python virtual environment

Now Runway can be used in the project via `pipenv run runway ...` (e.g. `pipenv run runway takeoff`).

To ensure future users of the project use the same version of Runway, direct them (e.g. via a Makefile) to invoke it via `pipenv sync; pipenv run Runway ...` – this will ensure the version in their virtual environment is kept in sync with the project's lock file.

Troubleshooting

Pipenv Not Found

If pipenv isn't available after installation (via `pip install --user pipenv`, see the python-setup guide.

2.1.4 Why Version Lock Per-Project

Locking the version of Runway per-project will allow you to:

- Specify the version(s) of Runway compatible with your deployments config
- Ensure Runway executions are performed with the same version (regardless of where/when they occur – avoids the dreaded "works on my machine")

2.2 Getting Started Guide

2.2.1 Basic Concepts

Welcome to Runway! To get a basic understanding of Runway, we have listed out the key concepts below that you will need to get started with deploying your first *module*.

Runway Config File

The *Runway config file* is usually stored at the root of a project repo. It defines the modules that will be managed by Runway.

Deployment

A *deployment* contains a list of modules and options for all the modules in the *deployment*. A *Runway config file* can contain multiple *deployments* and a *deployment* can contain multiple modules.

Module

A *module* is a directory containing a single infrastructure-as-code tool configuration of an application, a component, or some infrastructure (eg. a set of CloudFormation templates). It is defined in a *deployment* by path. Modules can also contain granular options that only pertain to it.

Environment

Environments are used for selecting the options/variables/parameters to be used with each modules. They can be defined by the name of a directory (if its not a git repo), git branch, or environment variable (`DEPLOY_ENVIRONMENT`). Standard environments would be something like prod, dev, and test.

No matter how the environment is determined, the name is made available to be consumed by your modules as the `DEPLOY_ENVIRONMENT` environment variable.

2.2.2 Deploying Your First Module

1. Create a directory for our project and change directory into the new directory.

```
# macOS example
$ mkdir sample-project
$ cd sample-project
```

2. Initialize the the new directory as a git repo and checkout branch **ENV-dev**. This will give us an environment of **dev**.

```
# macOS example
$ git init
$ git checkout -b ENV-dev
```

3. Download Runway using *curl*. Be sure to use the endpoint that corresponds to your operating system. Then, change the downloaded file's permissions to allow execution.

```
# macOS example
$ curl -L https://oni.ca/runway/latest/osx -o runway
$ chmod +x runway
```

4. Use Runway to generate a sample *module* using the *gen-sample* command. This will give us a preformatted *module* that is ready to be deployed after we change a few variables. To read more about the directory structure, see *Repo Structure*.

```
$ ./runway gen-sample cfn
```

5. To finish configuring our CloudFormation *module* , lets open the `dev-us-east-1.env` file that was created in `sampleapp.cfn/`. Here is where we will define values for our stacks that will be deployed as part of the **dev** environment in the **us-east-1** region. Replace the place holder values in this file with your own information. It is important that the `cfngin_bucket_name` value is globally unique for this example as it will be used to create a new S3 bucket.

```
namespace: onica-dev
customer: onica
environment: dev
region: us-east-1
# The CFNgin bucket is used for CFN template uploads to AWS
cfngin_bucket_name: cfngin-onica-us-east-1
```

6. With the *module* ready to deploy, now we need to create our *Runway config file*. Do to this, use the *init* command to generate a sample file at the root of the project repo.

```
$ ./runway init
```

runway.yml contents

```
---
# See full syntax at https://docs.onica.com/projects/runway/en/latest/
deployments:
  - modules:
      - nameofmyfirstmodulefolder
      - nameofmysecondmodulefolder
      # - etc...
    regions:
      - us-east-1
```

7. Now, we need to modify the `runway.yml` file that was just created to tell it where the *module* is located that we want it to deploy and what regions it will be deployed to. Each *module* type has their own configuration options which are described in more detail in the *Module Configurations* section but, for this example we are only concerned with the *CloudFormation module configuration*.

   The end result should like this:

```
---
# See full syntax at https://docs.onica.com/projects/runway/en/latest/
deployments:
  - modules:
      - sampleapp.cfn
    regions:
      - us-east-1
```

8. Before we deploy, it is always a good idea to know how the *module* will impact the currently deployed infrastructure in your AWS account. This is less of a concern for net-new infrastructure as it is when making modifications. But, for this example, lets run the *plan* command to see what is about to happen.

```
$ ./runway plan
```

9. We are finally ready to deploy! Use the *deploy* command to deploy our *module*.

```
$ ./runway deploy
```

We have only scratched the surface with what is possible in this example. Proceed below to find out how to delete the *module* we just deployed or, review the pages linked throughout this section to learn more about what we have done to this point before continuing.

2.2.3 Deleting Your First Module

From the root of the project directory we created in *Deploying Your First Module* we only need to run the *destroy* command to remove what we have deployed.

```
$ ./runway destroy
```

2.2.4 Execution Without A TTY (non-interactive)

Runway allows you to set an environment variable to allow execution without a TTY or if STDIN is closed. This allows users to execute Runway *deployments* in their CI/CD infrastructure as code deployment systems avoiding the `EOF when reading a line` error message. In order to execute runway without a TTY, set the `CI` environment variable before your `runway [deploy|destroy]` execution.

Important: Executing Runway in this way will cause Runway to perform updates in your environment without prompt. Use with caution.

2.3 Quickstart Guides

2.3.1 CloudFormation Quickstart

1. Prepare the project directory. See *Repo Structure* for more details.

```
mkdir my-app
cd my-app
git init
git checkout -b ENV-dev
```

2. Download/install Runway. Here we are showing the *curl* option. To see other available install methods, see *Installation*.

macOS

```
$ curl -L https://oni.ca/runway/latest/osx -o runway
$ chmod +x runway
```

Ubuntu

```
$ curl -L https://oni.ca/runway/latest/linux -o runway
$ chmod +x runway
```

Windows

```
> iwr -Uri oni.ca/runway/latest/windows -OutFile runway.exe
```

3. Use Runway to *generate a sample* CloudFormation *module*, edit the values in the environment file, and create a *Runway config file* to use the *module*.

macOS/Linux

```
$ runway gen-sample cfn
$ sed -i -e "s/CUSTOMERNAMEHERE/mydemo/g; s/ENVIRONMENTNAMEHERE/dev/g; s/stacker-/
↪stacker-$(uuidgen|tr "[:upper:]" "[:lower:]")-/g" sampleapp.cfn/dev-us-east-1.
↪env
$ cat <<EOF >> runway.yml
---
# Full syntax at https://github.com/onicagroup/runway
deployments:
  - modules:
      - sampleapp.cfn
    regions:
      - us-east-1
EOF
```

Windows

```
$ runway gen-sample cfn
$ (Get-Content sampleapp.cfn\dev-us-east-1.env).replace('CUSTOMERNAMEHERE',
↪'mydemo') | Set-Content sampleapp.cfn\dev-us-east-1.env
$ (Get-Content sampleapp.cfn\dev-us-east-1.env).replace('ENVIRONMENTNAMEHERE',
↪'dev') | Set-Content sampleapp.cfn\dev-us-east-1.env
$ (Get-Content sampleapp.cfn\dev-us-east-1.env).replace('stacker-', 'stacker-' +␣
↪[guid]::NewGuid() + '-') | Set-Content sampleapp.cfn\dev-us-east-1.env
$ $RunwayTemplate = @"
---
# Full syntax at https://github.com/onicagroup/runway
deployments:
  - modules:
      - sampleapp.cfn
    regions:
      - us-east-1
"@
$RunwayTemplate | Out-File -FilePath runway.yml -Encoding ASCII
```

4. *Deploy* the stack.

```
$ runway deploy
```

Now our stack is available at `mydemo-dev-sampleapp`, e.g.: `aws cloudformation describe-stack-resources --region us-east-1 --stack-name mydemo-dev-sampleapp`

2.3.2 Conduit (Serverless & CloudFront) Quickstart

Deploying the Conduit Web App

The Medium.com-clone "RealWorld" demo app named Conduit provides a simple demonstration of using Runway to deploy a Serverless Framework backend with an Angular frontend.

Prerequisites

- An AWS account, and configured terminal environment for interacting with it with an admin role.
- The following installed tools:
  - npm
  - yarn
  - git (Available out of the box on macOS)

Setup

1. Prepare the project directory. See *Repo Structure* for more details.

```
mkdir conduit
cd conduit
git init
git checkout -b ENV-dev
```

2. Download/install Runway. Here we are showing the *curl* option. To see other available install methods, see *Installation*.

macOS

```
curl -L https://oni.ca/runway/latest/osx -o runway
chmod +x runway
```

Ubuntu

```
curl -L https://oni.ca/runway/latest/linux -o runway
chmod +x runway
```

Windows

```
iwr -Uri oni.ca/runway/latest/windows -OutFile runway.exe
```

3. Download the source files.

macOS/Linux

```
curl -O https://codeload.github.com/anishkny/realworld-dynamodb-lambda/zip/v1.0.0
unzip v1.0.0
rm v1.0.0
mv realworld-dynamodb-lambda-1.0.0 backend
cd backend
sed -i '/package-lock\.json/d' .gitignore
echo '.dynamodb' >> .gitignore
npm install
cd ..
curl -O https://codeload.github.com/gothinkster/angular-realworld-example-app/zip/
↪35a66d144d8def340278cd55080d5c745714aca4
unzip 35a66d144d8def340278cd55080d5c745714aca4
rm 35a66d144d8def340278cd55080d5c745714aca4
mv angular-realworld-example-app-35a66d144d8def340278cd55080d5c745714aca4 frontend
cd frontend
mkdir scripts
cd scripts && { curl -O https://raw.githubusercontent.com/onicagroup/runway/
↪master/quickstarts/conduit/build.js ; cd -; }
sed -i 's/^\s*"build":\s.*$/    "build": "node scripts\/build",/' package.json
sed -i 's/^\s*"rxjs":\s.*$/    "rxjs": "~6.3.3",/' package.json
npm install
```

(continues on next page)

(continued from previous page)

```
curl -O https://raw.githubusercontent.com/onicagroup/runway/master/quickstarts/
↪conduit/update_env_endpoint.py
cd ..
curl -O https://raw.githubusercontent.com/onicagroup/runway/master/quickstarts/
↪conduit/runway.yml
```

Windows

```
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12
Invoke-WebRequest https://codeload.github.com/anishkny/realworld-dynamodb-lambda/
↪zip/v1.0.0 -OutFile v1.0.0.zip
Expand-Archive v1.0.0.zip .
Remove-Item v1.0.0.zip -Force
Rename-Item realworld-dynamodb-lambda-1.0.0 backend
cd backend
(gc .\.gitignore -raw).Replace("package-lock.json`n", "") | sc .\.gitignore
".dynamodb`r`n" | Out-File .\.gitignore -Append -Encoding UTF8
$(gc .\package.json) -replace "dynamodb install .*$", "dynamodb install`"" | Out-
↪File .\package.json -Force -Encoding UTF8
npm install
cd ..
Invoke-WebRequest https://codeload.github.com/gothinkster/angular-realworld-
↪example-app/zip/35a66d144d8def340278cd55080d5c745714aca4 -OutFile␣
↪35a66d144d8def340278cd55080d5c745714aca4.zip
Expand-Archive 35a66d144d8def340278cd55080d5c745714aca4.zip .
Remove-Item 35a66d144d8def340278cd55080d5c745714aca4.zip -Force
Rename-Item angular-realworld-example-app-
↪35a66d144d8def340278cd55080d5c745714aca4 frontend
cd frontend
(gc .\package.json -raw).Replace("`"rxjs`": `"^6.2.1`"", "`"rxjs`": `"~6.3.3`"")␣
↪| sc .\package.json
mkdir scripts
Invoke-WebRequest https://raw.githubusercontent.com/onicagroup/runway/master/
↪quickstarts/conduit/build.js -OutFile scripts/build.js
$(gc .\package.json) -replace "^\s*`"build`":\s.*$", "    `"build`": `"node␣
↪scripts/build`"," | Out-File .\package.json -Force -Encoding UTF8
npm install
Invoke-WebRequest https://raw.githubusercontent.com/onicagroup/runway/master/
↪quickstarts/conduit/update_env_endpoint.py -OutFile update_env_endpoint.py
cd ..
Invoke-WebRequest https://raw.githubusercontent.com/onicagroup/runway/master/
↪quickstarts/conduit/Pipfile -OutFile Pipfile
Invoke-WebRequest https://raw.githubusercontent.com/onicagroup/runway/master/
↪quickstarts/conduit/runway.yml -OutFile runway.yml
```

Deploying

Execute `pipenv run runway deploy`, enter `all` (to deploy the backend followed by the frontend). Deployment will take some time (mostly waiting for the CloudFront distribution to stabilize).

The CloudFront domain at which the site can be reached will be displayed near the last lines of output once deployment is complete, e.g.:

```
staticsite:  sync & CF invalidation of E17B5JWPMTX5Z8 (domain ddy1q4je03d7u.
cloudfront.net) complete
```

Teardown

Execute `pipenv run runway destroy`, enter `all`.

The backend DynamoDB tables will still be retained after the destroy is complete. They must be deleted separately:

On macOS/Linux:

```
for i in realworld-dev-articles realworld-dev-comments realworld-dev-users; do aws␣
↪dynamodb delete-table --region us-east-1 --table-name $i; done
```

On Windows:

```
foreach($table in @("realworld-dev-articles", "realworld-dev-comments", "realworld-
↪dev-users"))
{
  CMD /C "pipenv run aws dynamodb delete-table --region us-east-1 --table-name $table"
}
```

Next Steps / Additional Notes

The serverless-plugin-export-endpoints plugin is a good alternative to the custom update_env_endpoint.py script deployed above to update the environment file.

Permissions

The specific IAM permissions required to manage the resources in this demo are as follows

```
# CloudFormation
- cloudformation:CreateStack
- cloudformation:DeleteStack
- cloudformation:CreateChangeSet
- cloudformation:DescribeChangeSet
- cloudformation:DeleteChangeSet
- cloudformation:DescribeStackResource
- cloudformation:DescribeStackResources
- cloudformation:DescribeStacks
- cloudformation:DescribeStackEvents
- cloudformation:GetTemplate
- cloudformation:UpdateStack
- cloudformation:ExecuteChangeSet
- cloudformation:ValidateTemplate
# Serverless
- apigateway:GET
```

(continues on next page)

(continued from previous page)

```
- apigateway:DELETE
- apigateway:POST
- apigateway:PUT
- lambda:AddPermission
- lambda:CreateAlias
- lambda:CreateFunction
- lambda:DeleteAlias
- lambda:DeleteFunction
- lambda:GetFunction
- lambda:GetFunctionConfiguration
- lambda:ListVersionsByFunction
- lambda:PublishVersion
- lambda:UpdateAlias
- lambda:UpdateFunctionCode
- lambda:UpdateFunctionConfiguration
- iam:CreateRole
- iam:DeleteRole
- iam:DeleteRolePolicy
- iam:GetRole
- iam:PassRole
- iam:PutRolePolicy
- logs:CreateLogGroup
- logs:DeleteLogGroup
- logs:DescribeLogGroups
- s3:CreateBucket
- s3:DeleteBucket
- s3:DeleteBucketPolicy
- s3:DeleteObject
- s3:DeleteObjectVersion
- s3:GetObjectVersion
- s3:ListBucket
- s3:ListBucketVersions
- s3:PutBucketVersioning
- s3:PutBucketPolicy
- s3:PutLifecycleConfiguration
# Frontend
- cloudfront:CreateCloudFrontOriginAccessIdentity
- cloudfront:CreateDistribution
- cloudfront:CreateInvalidation
- cloudfront:DeleteCloudFrontOriginAccessIdentity
- cloudfront:DeleteDistribution
- cloudfront:GetCloudFrontOriginAccessIdentity
- cloudfront:GetCloudFrontOriginAccessIdentityConfig
- cloudfront:GetDistribution
- cloudfront:GetDistributionConfig
- cloudfront:GetInvalidation
- cloudfront:ListDistributions
- cloudfront:TagResource
- cloudfront:UntagResource
- cloudfront:UpdateCloudFrontOriginAccessIdentity
- cloudfront:UpdateDistribution
- s3:DeleteBucketWebsite
- s3:GetBucketAcl
- s3:GetObject
- s3:PutBucketAcl
- s3:GetBucketWebsite
- s3:PutBucketWebsite
```

(continues on next page)

(continued from previous page)

```
- s3:PutObject
- ssm:GetParameter
- ssm:PutParameter
# Backend
- dynamodb:CreateTable
- dynamodb:DeleteTable
- dynamodb:DescribeTable
- dynamodb:TagResource
- dynamodb:UntagResource
- dynamodb:UpdateTable
```

2.3.3 Other Ways to Use Runway

While we recommend using one of the install methods outlined in the *Installation* section, we realize that these may not be an option for some so we have provided a *CloudFormation* template for spinning up a deploy environment in AWS and a *Docker* image/Dockerfile that can be used to run Runway.

CloudFormation

This CloudFormation template is probably the easiest and quickest way to go from "zero to Runway" as it allows for using an IAM Role eliminate the need to configure API keys. The template will deploy your preference of Linux or Windows Runway host. Windows Runway host includes vsCode, which some users may find easier for manipulating Runway config files.

Docker

Docker users can build their own Docker image to run a local Runway container or modify this Dockerfile to build a Runway image to suit specific needs.

We have also provide a pre-build Docker image on Docker Hub that can be used with the following command.

```
$ docker run -it --rm onica/runway-quickstart
```

2.4 Commands

2.4.1 deploy

Used to deploy *modules* with Runway.

When run, the environment is determined from the current git branch unless `ignore_git_branch: true` is specified in the *Runway config file*. If the `DEPLOY_ENVIRONMENT` environment variable is set, it's value will be used. If neither the git branch or environment variable are available, the directory name is used. The environment identified here is used to determine the env/config files to use. It is also used with options defined in the Runway config file such as `assume_role`, `account_id`, etc. See *Runway Config* for details on these options.

The user will be prompted to select which *deployment(s)* and *module(s)* to process unless there is only one *deployment* and/or *module*, the environment variable `CI` is set, or the `--tag <tag>...` option provided is used. In which case, the *deployment(s)* and *module(s)* will be processed in sequence, in the order they are defined.

Options

`--tag <tag>...`	Select modules for processing by tag or tags. This option can be specified more than once to build a list of tags that are treated as "AND". (ex. `--tag <tag1> --tag <tag2>` would select all modules with BOTH tags).

Example

```
# manually select deployment(s) and module(s)
$ runway deploy

# select all modules with the tag 'app:example' AND 'my-tag'
$ runway deploy --tag app:example --tag my-tag

# process all deployment(s) and module(s)
$ CI=1 runway deploy
```

2.4.2 destroy

Used to destroy *modules* with Runway.

Danger: Use extreme caution when using with `CI` or `--tag <tag>...`. You will **not** be prompted before deletion. All modules (or those selected by tag) for the current environment will be **irrecoverably deleted**.

When run, the environment is determined from the current git branch unless `ignore_git_branch: true` is specified in the *Runway config file*. If the `DEPLOY_ENVIRONMENT` environment variable is set, it's value will be used. If neither the git branch or environment variable are available, the directory name is used. The environment identified here is used to determine the env/config files to use. It is also used with options defined in the Runway config file such as `assume_role`, `account_id`, etc. See *Runway Config* for details on these options.

The user will be prompted to select which *deployment(s)* and *module(s)* to process unless there is only one *deployment* and/or *module*, the environment variable `CI` is set, or the `--tag <tag>...` option provided is used. In which case, the *deployment(s)* and *module(s)* will be processed in sequence, in reverse of the order they are defined.

Options

`--tag <tag>...`	Select modules for processing by tag or tags. This option can be specified more than once to build a list of tags that are treated as "AND". (ex. `--tag <tag1> --tag <tag2>` would select all modules with BOTH tags).

Example

```
# manually select deployment(s) and module(s)
$ runway destroy

# select all modules with the tag 'app:example' AND 'my-tag'
$ runway destroy --tag app:example --tag my-tag

# process all deployment(s) and module(s)
$ CI=1 runway destroy
```

2.4.3 dismantle

Alias of *destroy*.

2.4.4 envvars

Output `runway.yml`-defined environment variables.

OS environment variables can be set in `runway.yml` for different Runway environments (e.g. dev & prod `KUBECONFIG` values). The `envvars` command allows access to these values for use outside of Runway.

Example

```
$ eval "$(runway envvars)"
```

2.4.5 gen-sample

Generate a sample *Runway module* directory.

The sample module is created in the current directory. If a directory already exists with the name it tries to use, it will not create the sample directory.

Available Samples

Name	Description
`cdk-csharp`	AWS CDK *module* using C#
`cdk-py`	AWS CDK *module* using Python
`cdk-tsc`	AWS CDK *module* using TypeScript
`cfn`	CloudFormation *module* stack with S3 bucket & DDB table (perfect for storing Terraform backend state)
`k8s-cfn-repo`	Kubernetes *module* EKS cluster & sample app using CloudFormation
`k8s-tf-repo`	Kubernetes *module* EKS cluster & sample app using Terraform
`sls-py`	Serverless Framework *module* using Python
`sls-tsc`	Serverless Framework *module* using TypeScript
`stacker`	Troposphere/Stacker *module* identical the `cfn` sample but with the template written in python
`static-angular`	**`StaticSite`_** *module* of a StaticSite and the Angular framework
`static-react`	**`StaticSite`_** *module* of a StaticSite and the React framework
`tf`	Terraform *module*

Example

```
# create a "sampleapp.cfn" sample module directory
$ runway gen-sample cfn

# create a "runway-sample-tfstate.cfn" sample module directory
$ runway gen-sample stacker

# create a "sampleapp.sls" sample module directory
$ runway gen-sample sls-py
```

2.4.6 init

Creates a sample *Runway Config File* in the current directory.

If a *Runway config file* is already present, no action is taken.

Example

```
$ runway init
```

Sample Runway Config File

```
---
# See full syntax at https://docs.onica.com/projects/runway/en/latest/
deployments:
  - modules:
      - nameofmyfirstmodulefolder
      - nameofmysecondmodulefolder
      # - etc...
```

(continues on next page)

(continued from previous page)

```
    regions:
      - us-east-1
```

2.4.7 kbenv

Manage versions and execute Kubernetes commands.

Runway's built-in kubectl version management ensure the correct version is used for a given environment. Define a `.kubectl-version` file in your k8s module and that version will be automatically downloaded & used during Runway operations.

The `tfenv` subcommand supplements this functionality in 2 ways:

- The `install` option will download kubectl (e.g. for pre-seeding a deployment system)
- The `run` option will execute arbitrary kubectl commands

Examples

```
$ runway kbenv install 1.14.5
$ runway kbenv install  # retrieves version from .kubectl-version

$ runway kbenv run -- get namespace
```

2.4.8 plan

Used to plan actions by comparing what is live and what is defined locally.

Note: Currently only supported for AWS CDK, CloudFormation, Terraform, and Troposphere.

When run, the environment is determined from the current git branch unless `ignore_git_branch: true` is specified in the *Runway config file*. If the `DEPLOY_ENVIRONMENT` environment variable is set, it's value will be used. If neither the git branch or environment variable are available, the directory name is used. The environment identified here is used to determine the env/config files to use. It is also used with options defined in the Runway config file such as `assume_role`, `account_id`, etc. See *Runway Config* for details on these options.

The user will be prompted to select which *deployment(s)* and *module(s)* to process unless there is only one *deployment* and/or *module*, the environment variable `CI` is set, or the `--tag <tag>...` option provided is used. In which case, the *deployment(s)* and *module(s)* will be processed in sequence, in the order they are defined.

Options

`--tag <tag>...`	Select modules for processing by tag or tags. This option can be specified more than once to build a list of tags that are treated as "AND". (ex. `--tag <tag1> --tag <tag2>` would select all modules with BOTH tags).

Equivalent To

There are the native commands that are used:

- `cdk diff` - https://docs.aws.amazon.com/cdk/latest/guide/tools.html
- `stacker diff` - https://stacker.readthedocs.io/en/stable/commands.html#diff
- `terraform plan` - https://www.terraform.io/docs/commands/plan.html

Example

```
$ runway plan
```

2.4.9 preflight

Alias of *test*.

2.4.10 run-aws

Execute aws cli commands using the version bundled with Runway.

This command gives access to the `aws` CLI when it might not otherwise be installed (e.g. when using the bundled version of Runway).

Example

```
$ runway run-aws -- s3 ls
```

2.4.11 run-python

Execute a python script using a bundled copy of python.

By using this command Runway can execute actions using a bundled copy of python without requiring python to be installed on a system. This is only applicable when installing the bundled version of Runway, not from PyPI (`pip install runway`). When installed from PyPI, the system's python is used.

Example

```
$ runway run-python my-script.py
```

2.4.12 run-stacker

Execute the "shimmed" Stacker aka Runway CFNgin.

This command allows direct access to Runway's CloudFormation management tool.

Example

```
$ runway run-stacker -- build example.env example.yaml
```

2.4.13 takeoff

Alias of *deploy*.

2.4.14 taxi

Alias of *plan*.

2.4.15 test

Execute *tests* as defined in the *Runway Config File*.

If one of the tests fails, the command will exit unless the `required` option is set to `false` for the failing test. If it is not required, the next test will be executed.

References

- *Runway Config File/Test*
- *Defining Tests*

2.4.16 tfenv

Manage versions and execute Terraform commands.

Runway's built-in Terraform version management allows for long-term stability of Terraform executions. Define a `.terraform-version` file in your Terraform module and that version will be automatically downloaded & used during Runway operations.

The `tfenv` subcommand supplements this functionality in 2 ways:

- The `install` option will download Terraform (e.g. for pre-seeding a deployment system)
- The `run` option will execute arbitrary Terraform commands

Examples

```
$ runway tfenv install 0.12.1
$ runway tfenv install  # retrieves version from .terraform-version

$ runway tfenv run -- workspace list
```

2.4.17 whichenv

Identify the current environment and print it to the terminal.

When run, the environment is determined from the current git branch unless `ignore_git_branch: true` is specified in the *Runway config file*. If the `DEPLOY_ENVIRONMENT` environment variable is set, it's value will be used. If neither the git branch or environment variable are available, the directory name is used. The environment identified here is used to determine the env/config files to use. It is also used with options defined in the Runway config file such as `assume_role`, `account_id`, etc. See *Runway Config* for details on these options.

Example

```
$ runway whichenv
common
```

2.5 Runway Config File

2.5.1 Top-Level Configuration

class runway.config.**Config**(*deployments*, *tests=None*, *ignore_git_branch=False*, *variables=None*)

The Runway config file is where all options are defined.

It contains definitions for deployments, tests, and some global options that impact core functionality.

The Runway config file can have two possible names, `runway.yml` or `runway.yaml`. It must be stored at the root of the directory containing all modules to be deployed.

Example

```
---
# See full syntax at https://github.com/onicagroup/runway
ignore_git_branch: true
tests:
  - name: example
    type: script
    args:
      commands:
        - echo "Hello world"
deployments:
  - modules:
      - path: my-modules.cfn
    regions:
      - us-east-1
```

Keyword Arguments

- **deployments** (*List[Dict[str, Any]]*) – A list of *deployments* that are processed in the order they are defined.
- **tests** (*Optional[List[Dict[str, Any]]]*) – A list of *tests* that are processed in the order they are defined.
- **ignore_git_branch** (*bool*) – Disable git branch lookup when using environment folders, non-git VCS, or defining the DEPLOY_ENVIRONMENT environment variable before execution. Note that defining DEPLOY_ENVIRONMENT will automatically ignore the git branch.
- **variables** (*Optional[Dict[str, Any]]*) – A map that defines the location of a variables file and/or the variables themselves.

Lookup Resolution

Keyword / Directive	Support
deployments	No direct support. See *Deployment* for details on support within a deploymet definition.
tests	No direct support. See *Test* for details on support within a test definition.
ignore_git_branch	None
variables	None

References

- *deployment*
- *test*

2.5.2 Deployment

class runway.config.**DeploymentDefinition**(*deployment*)

A deployment defines modules and options that affect the modules.

Deployments are processed during a `deploy`/`destroy`/`plan` action. If the processing of one deployment fails, the action will end.

During a `deploy`/`destroy` action, the user has the option to select which deployment will run unless the `CI` environment variable is set, the `--tag <tag>...` cli option was provided, or only one deployment is defined.

Example

```yaml
deployments:
  - modules:  # minimum requirements for a deployment
      # "./" can alternatively be used for the module name to indicate
      # the current directory
      - my-module.cfn
    regions:
      - us-east-1
  - name: detailed-deployment  # optional
    modules:
      - path: my-other-modules.cfn
        type: cloudformation
    regions:
      - us-east-1
    environments:
      prod: 111111111111/us-east-1
      dev:
        - 222222222222/us-east-1
        - 333333333333/us-east-1
      lab: true
    account_id: ${var account_ids}  # optional
    assume_role: ${var assume_role}  # optional
    parameters:  # optional
        region: ${env AWS_REGION}
        image_id: ${var image_id.${env DEPLOY_ENVIRONMENT}}
    env_vars:  # optional environment variable overrides
        AWS_PROFILE: ${var aws_profile.${env DEPLOY_ENVIRONMENT}::default=default}
        APP_PATH: ${var app_path.${env DEPLOY_ENVIRONMENT}}
```

Keyword Arguments

- **account_alias** (*Optional[Dict[str, str]]*) – A mapping of `$environment: $alias` that, if provided, is used to verify the currently assumed role or credentials.
- **account_id** (*Optional[Dict[str, Union[str, int]]]*) – A mapping of `$environment: $id` that, if provided, is used to verify the currently assumed role or credentials.
- **assume_role**(*Optional[Dict[str, Union[str, Dict[str, str]]]]*)– A mapping of `$environment: $role` or `$environment: {arn: $role, duration: $int}` to assume a role when processing a deployment. `arn: $role` can be used to apply the same role to all environment. `post_deploy_env_revert: true` can also be provided to revert credentials after processing.

- **environments** (*Optional[Dict[str, Dict[str, Any]]]*) – Optional mapping of environment names to a booleon value used to explicitly enable or disable in an environment. This can be used when an environment specific variables file and parameters are not needed to force a module to enable anyway or, explicitly skip a module even if a file or parameters are found. The mapping can also have a string (or list of strings) value of $ACCOUNT_ID/$REGION to lock an environment to specific regions in a specific accounts. If it matches, it will act as an explicit enable.
- **env_vars** (*Optional[Dict[str, Dict[str, Any]]]*) – A mapping of OS environment variable overrides to apply when processing modules in the deployment. Can be defined per environment or for all environments by omiting the environment name.
- **modules** (*Optional[List[Dict[str, Any]]]*) – A list of modules to be processed in the order they are defined.
- **module_options** (*Optional[Dict[str, Any]]*) – Options that are shared among all modules in the deployment.
- **name** (*str*) – Name of the deployment. Used to more easily identify where different deployments begin/end in the logs.
- **type** (*str*) – The type of module we are deploying. By default Runway will first check to see if you explicitly specify the module type, after that it will check to see if a valid module extension exists on the directory, and finally it will attempt to autodetect the type of module. Valid values are: `serverless`, `terraform`, `cdk`, `kubernetes`, `cloudformation`, `static`.
- **regions** (*List[str]*) – AWS region names where modules will be deployed/destroyed. Can optionally define as a map with `parallel` as the key and a list of regions as the value. See **parallel_regions** for more info.
- **parallel_regions** – Can be defined in place of `regions.parallel[]`. This will cause all modules in the deployment to be executed in all provided regions in parallel (at the same time). Only takes effect when the `CI` environment variable is set, enabling non-interactive mode, as prompts will not be able to be presented. If `CI` is not set, the regions will be processed one at a time. This can be used in tandom with **parallel modules**. `assume_role.post_deploy_env_revert` will always be `true` when run in parallel.
- **parameters** (*Optional(Dict[str, Any])*) – Module level parameters that are akin to a CloudFormation parameter in functionality. These can be used to pass variable values to your modules in place of a `.env`/`.tfenv`/environment config file. Through the use of Lookups, the value can differ per deploy environment, region, etc.

Lookup Resolution

Important: Due to how a deployment is processed, values are resolved twice. Once before processing and once during processing. Because of this, the keywords/directives that are resolved before processing will not have access to values set during process like `AWS_REGION`, `AWS_DEFAULT_REGION`, and `DEPLOY_ENVIRONMENT` for the pre-processing resolution but, if they are resolved again during processing, these will be available. To avoide errors during the first resolution due to the value not existing, provide a default value for the *Lookup*.

Keyword / Directive	Support
account_alias	env lookup (AWS_REGION and AWS_DEFAULT_REGION will not have been set by Runway yet), var lookup
account_id	env lookup (AWS_REGION and AWS_DEFAULT_REGION will not have been set by Runway yet), var lookup
assume_role	env lookup (AWS_REGION and AWS_DEFAULT_REGION will not have been set by Runway yet), var lookup
environments	env lookup, var lookup
env_vars	env lookup (AWS_REGION, DEPLOY_ENVIRONMENT, and AWS_DEFAULT_REGION will not have been set by Runway during pre-process resolution. provide a default value to avoide errors.), var lookup
modules	No direct support. See *module* for details on support within a module definition.
module_options	env lookup, var lookup
name	None
regions	env lookup (AWS_REGION and AWS_DEFAULT_REGION will not have been set by Runway yet), var lookup
parallel_regions	env lookup (AWS_REGION and AWS_DEFAULT_REGION will not have been set by Runway yet), var lookup
parameters	env lookup, var lookup

References

- *module*
- *deploy*
- *destroy*
- *plan*

2.5.3 Module

class runway.config.**ModuleDefinition**(*name*, *path*, *class_path=None*, *type_str=None*, *environments=None*, *parameters=None*, *env_vars=None*, *options=None*, *tags=None*, *child_modules=None*)

A module defines the directory to be processed and applicable options.

It can consist of CloudFormation (using CFNgin), Terraform, Serverless Framework, AWS CDK, Kubernetes, or a *Static Site*. It is recommended to place the appropriate extension on each directory for identification (but it is not required). See *Repo Structure* for examples of a module directory structure.

Suffix/Extension	IaC Tool/Framework
.cdk	AWS CDK
.cfn	CloudFormation
.sls	Serverless Framework
.tf	Terraform
.k8s	Kubernetes
.web	*Static Site*

A module is only deployed if there is a corresponding environment file present or parameters are provided. This can take the form of either a file in the module folder or the parameters option being defined. The naming format varies per-module type. See *Module Configurations* for acceptable environment file name formats.

Modules can be defined as a string or a mapping. The minimum requirement for a module is a string that is equal to the name of the module directory. Providing a string is the same as providing a value for `path` in a mapping definition.

Example

```yaml
deployments:
  - modules:
      - my-module.cfn  # this
      - path: my-module.cfn  # is the same as this
```

Using a map to define a module provides the ability to specify per-module `options`, parameters, environment variables,tags, and even a custom class for processing the module. The options that can be used with each module vary. For detailed information about module-specific options, see *Module Configurations*.

Example

```yaml
deployments:
  - modules:
      - name: my-module
        path: my-module.tf
        environments:
          prod: 111111111111/us-east-1
          dev:
            - 222222222222/us-east-1
            - 333333333333/us-east-1
          lab: true
        parameters:
          image_id: ${var image_id.${env DEPLOY_ENVIRONMENT}}
        tags:
          - app:example
          - my-tag
        options:
          terraform_backend_config:
            region: us-east-1
          terraform_backend_cfn_outputs:
            bucket: StackName::OutputName
            dynamodb_table: StackName::OutputName
```

One special map keyword, `parallel`, indicates a list of child modules that will be executed in parallel (simultaneously) if the `CI` *environment variable is set*.

Example

In this example, `backend.tf` will be deployed followed by the services that will be utilizing it. The services will be deployed in parallel. After the services have completed, `frontend.tf` will be deployed.

```yaml
deployments:
  - modules:
    - backend.tf
    - parallel:
      - servicea.cfn  # any normal module option can be used here
      - path: serviceb.cfn
```

(continues on next page)

(continued from previous page)

```
    - path: servicec.cfn
      parameters:
        count: ${var count.${env DEPLOY_ENVIRONMENT}}
  - frontend.tf
```

Keyword Arguments

- **name** (*str*) – Name of the module. Used to more easily identify where different modules begin/end in the logs.
- **path** (*str*) – Path to the module relative to the Runway config file. This cannot be higher than the Runway config file. See *Path* for detailed usage.
- **class_path** (*Optional[str]*) – Path to custom Runway module class. Also used for static site deployments. See *Module Configurations* for detailed usage.
- **type_str** (*Optional[str]*) – Alias for type of module to use *Module Configurations* for detailed usage.
- **environments** (*Optional[Dict[str, Dict[str, Any]]]*) – Optional mapping of environment names to a booleon value used to explicitly deploy or not deploy in an environment. This can be used when an environment specific variables file and parameters are not needed to force a module to deploy anyway or, explicitly skip a module even if a file or parameters are found. The mapping can also have a string (or list of strings) value of $ACCOUNT_ID/$REGION to lock an environment to specific regions in a specific accounts. If it matches, it will act as an explicit deploy.
- **env_vars** (*Optional[Dict[str, Dict[str, Any]]]*) – A mapping of OS environment variable overrides to apply when processing modules in the deployment. Can be defined per environment or for all environments by omiting the environment name. Takes precedence over values set at the deployment-level.
- **options** (*Optional[Dict[str, Any]]*) – Module-specific options. See *Module Configurations* for detailed usage. Takes precedence over values set at the deployment-level.
- **parameters** (*Optional(Dict[str, Any])*) – Module level parameters that are akin to a CloudFormation parameter in functionality. These can be used to pass variable values to your modules in place of a `.env`/`.tfenv`/environment config file. Through the use of Lookups, the value can differ per deploy environment, region, etc.
- **tags** (*Optional[Dict[str, str]]*) – Module tags used to select which modules to process using CLI arguments. (`--tag <tag>...`)
- **child_modules** (*Optional[List[Union[str, Dict[str, Any]]]]*) – Child modules that can be executed in parallel

Lookup Resolution

Keyword / Directive	Support
`name`	None
`path`	env lookup, var lookup
`class_path`	env lookup, var lookup
`environments`	env lookup, var lookup
`env_vars`	env lookup, var lookup
`options`	env lookup, var lookup
`parameters`	env lookup, var lookup
`tags`	None

References

- AWS CDK
- CloudFormation
- Serverless Framework
- CFNgin
- Troposphere
- Terraform
- Kubernetes
- *Static Site*
- *Module Configurations* - detailed module `options`
- *Repo Structure* - examples of directory structure
- *deploy*
- *destroy*
- *plan*

Path

Runway configuration `path` settings object.

Path is responsible for parsing the `path` property of a Runway configuration. It then can determine if the path specified is locally sourced or remotely sourced through a service such as *Git* or S3.

Local `path` variables are defined relative to the root project folder. The value for this cannot be higher than the Runway config file, it must be at the runway file itself or in a sub directory.

Example

```
deployments:
- modules:
    - path: my/local/module.cfn
    - my/local/module.cfn # same as above
    - ./ # module is in the root
```

When the `path` is remote, Runway is responsible for fetching the resource and returning the location of it's cached path. The information for retrieving those sources can be controlled via runway rather than manually retrieving each one.

Example

```
deployments:
- modules:
    - path: git::git://github.com/your_handle/your_repo.git//my-module.cfn
```

The `path` structure is based on the encoding found in Terraform modules.

The values parsed from the string are as follows:

source

Determine if the source is local or remote. The initial prefix is used to determine this separated by *::* in the string. A path is considered local if it contains no source type value.

Example

```
deployments:
    - modules:
        # source is `git`
        - path: git::git://github.com/foo/bar.git
```

uri

The uniform resource identifier when targetting a remote resource. This instructs runway on where to retrieve your module.

Example

```
deployments:
    - modules:
        # uri is `git://github.com/foo/bar.git`
        - path: git::git://github.com/foo/bar.git
```

location

The relative location of the module files from the root directory. This value is specified as a path after the uri separated by *//*

Example

```
deployments:
    - modules:
        # location is `my/path`
        - path: git::git://github.com/foo/bar.git//my/path
```

options

The remaining options that are passed along to the Source. This is specified in the path following the *?* separator. Multiple option keys and values can be specified with the *&* as the separator. Each remote source can have different options for retrieval, please make sure to review individual source types to get more information on properly formatting.

Example

```
deployments:
    - modules:
        # options are `foo=bar&ba=bop`
        - path: git::git://github.com/foo/bar.git//my/path?foo=bar&baz=bop
```

Git

Git remote resources can be used as modules for your Runway project. Below is an example of git remote path.

Example:

```
deployments:
    - modules:
        - git::git://github.com/foo/bar.git//my/path?branch=develop
```

The path is broken down into the following attributes:

`git`: The type of remote resource being retrieved, in this case **git**

`::`: Logical separator of the type from the rest of the path string

`git://github.com/foo/bar.git`: The protocol and URI address of the git repository

`//` **(optional)**: Logical separator of the URI from the remaining path string

`my/path` **(optional)**: The relative path from the root of the repo where the module is housed

`?` **(optional)**: Logical separator of the path from the options

`branch=develop` **(optional)**: The options to be passed. The Git module accepts three different types of options: *commit*, *tag*, or *branch*. These respectively point the repository at the reference id specified.

Type

Runway configuration `type` settings object.

The `type` property of a Runway configuration can be used to explicitly specify what module type you are intending to deploy.

Runway determines the type of module you are trying to deploy in 3 different ways. First, it will check for the `type` property as described here, next it will look for a suffix as described in Module Definition, and finally it will attempt to autodetect your module type by scanning the files of the project. If none of those settings produces a valid result an error will occur. The following are valid explicit types:

Type	IaC Tool/Framework
`cdk`	AWS CDK
`cloudformation`	CloudFormation
`serverless`	Serverless Framework
`terraform`	Terraform
`kubernetes`	Kubernetes
`static`	*Static Site*

Even when specifying a module `type` the module structure needs to be conducive with that type of project. If the files contained within don't match the type then an error will occur.

2.5.4 Test

class `runway.config.`**TestDefinition**(*name*, *test_type*, *args=None*, *required=True*)

Tests can be defined as part of the Runway config file.

This is to remove the need for complex Makefiles or scripts to initiate test runners. Simply define all tests for a project in the Runway config file and use the `runway test` *command* to execute them.

Example

```
tests:
  - name: my-test
    type: script
    required: false
    args:
      commands:
        - echo "Hello World!"
```

Keyword Arguments

- **name** (*str*) – Name of the test. Used to more easily identify where different tests begin/end in the logs.
- **type** (*str*) – The type of test to run. See *Build-in Test Types* for supported test types.
- **args** (*Optional[Dict[str, Any]]*) – Arguments to be passed to the test. Supported arguments vary by test type. See *Build-in Test Types* for the list of arguments supported by each test type.
- **required** (*bool*) – If false, testing will continue if the test fails. *(default: true)*

Lookup Resolution

Note: Runway does not set `AWS_REGION` or `AWS_DEFAULT_REGION` environment variables. If the `DEPLOY_ENVIRONMENT` environment variable is not manually set, it will always be `test` and is not determined from the branch or directory.

Keyword / Directive	Support
`args`	env lookup, var lookup
`required`	env lookup, var lookup

References

- *Build-in Test Types* - Supported test types and their arguments
- *test command*

2.5.5 Variables

class runway.config.**VariablesDefinition**(*file_path=None*, *sys_path=None*, ***kwargs*)

A variable definitions for the Runway config file.

Runway variables are used to fill values that could change based on any number of circumstances. They can also be used to simplify the Runway config file by pulling lengthy definitions into another file. Variables can be used in the config file by providing the var lookup to any keyword/directive that supports *Lookups*.

By default, Runway will look for and load a `runway.variables.yml` or `runway.variables.yaml` file that is in the same directory as the Runway config file. The file path and name of the file can optionally be defined in the config file. If the file path is explicitly provided and the file can't be found, an error will be raised.

Variables can also be defined in the Runway config file directly. This can either be in place of a dedicated variables file, extend an existing file, or override values from the file.

Lookup Resolution

Runway lookup resolution is not supported within the variables definition block or variables file. Attempts to use Runway *Lookups* within the variables definition block or variables file will result in the literal value being processed.

Example

```yaml
variables:
  sys_path: ./  # defaults to the current directory
  file_path: secrets.yaml
  # define additional variables or override those in the variables file
  another_var: some_value
deployments:
  - modules:
      - ${var sampleapp.definition}
    regions: ${var sampleapp.regions}
```

Keyword Arguments

- **file_path** – Explicit path to a variables file. If it cannot be found Runway will exit.
- **sys_path** – Directory to base relative paths off of.

2.5.6 Sample

runway.yml

```yaml
---
# Order that tests will be run. Test execution is triggered with the
# 'runway test' command. Testing will fail and exit if any of the
# individual tests fail unless they are marked with 'required: false'.
# Please see the doc section dedicated to tests for more details.

tests:
  - name: test-names-are-optional
    type: script  # there are a few built in test types
    args:  # each test has their own set of arguments they can accept
      commands:
        - echo "Beginning a test..."
        - cd app.sls && npm test && cd ..
        - echo "Test complete!"
  - name: unimportant-test
    type: cfn-lint
    required: false  # tests will still pass if this fails
  - type: yamllint  # not all tests accept arguments

# Order that modules will be deployed. A module will be skipped if a
# corresponding environment file is not present or "enabled" is false.
# E.g., for cfn modules, if
# 1) a dev-us-west-2.env file is not in the 'app.cfn' folder when running
#    a dev deployment of 'app' to us-west-2,
# and
# 2) "enabled" is false under the deployment or module
#
# then it will be skipped.

deployments:
  - modules:
      - myapp.cfn
    regions:
      - us-west-2

  - name: terraformapp  # deployments can optionally have names
    modules:
      - myapp.tf
    regions:
      - us-east-1
    assume_role:  # optional
      # When running multiple deployments, post_deploy_env_revert can be used
      # to revert the AWS credentials in the environment to their previous
      # values
      # post_deploy_env_revert: true
      arn: ${var assume_role.${env DEPLOY_ENVIRONMENT}}
      # duration: 7200
```

(continues on next page)

(continued from previous page)

```
    # Parameters (e.g. values for CFN .env file, TF .tfvars) can
    # be provided at the deployment level -- the options will be applied to
    # every module
    parameters:
      region: ${env AWS_REGION}
      image_id: ${var image_id.${env DEPLOY_ENVIRONMENT}}

    # AWS account alias can be provided to have Runway verify the current
    # assumed role / credentials match the necessary account
    account_alias: ${var account_alias.${env DEPLOY_ENVIRONMENT}}  # optional

    # AWS account id can be provided to have Runway verify the current
    # assumed role / credentials match the necessary account
    account_id: ${var account_id.${env DEPLOY_ENVIRONMENT}}  # optional

    # env_vars set OS environment variables for the module (not logical
    # environment values like those in a CFN .env or TF .tfvars file).
    # They should generally not be used (they are provided for use with
    # tools that absolutely require it, like Terraform's
    # TF_PLUGIN_CACHE_DIR option)
    env_vars:  # optional environment variable overrides
      AWS_PROFILE: ${var envvars.profile.${env DEPLOY_ENVIRONMENT}}
      APP_PATH: ${var envvars.app_path}
      ANOTHER_VAR: foo

  # Start of another deployment
  - modules:
      - path: myapp.cfn
        # Parameters (e.g. values for CFN .env file, TF .tfvars) can
        # be provided for a single module (replacing or supplementing the
        # use of environment/tfvars/etc files in the module)
        parameters:
          region: ${env AWS_REGION}
          image_id: ${var image_id.${env DEPLOY_ENVIRONMENT}}
        tags:  # Modules can optionally have tags.
          # This is a list of strings that can be "targeted"
          # by passing arguments to the deploy/destroy command.
          - some-string
          - app:example
          - tier:web
          - owner:onica
          # example: `runway deploy --tag app:example --tag tier:web`
          #   This would select any modules with BOTH app:example AND tier:web
    regions:
      - us-west-2

# If using environment folders instead of git branches, git branch lookup can
# be disabled entirely (see "Repo Structure")
# ignore_git_branch: true
```

variables.runway.yml

```yaml
account_alias:
  dev: my_dev_account
  prod: my_dev_account
account_id:
  dev: 123456789012
  prod: 345678901234
assume_role:
  dev: arn:aws:iam::account-id1:role/role-name
  prod: arn:aws:iam::account-id2:role/role-name
image_id:
  dev: ami-abc123
envvars:
  profile:
    dev: foo
    prod: bar
  app_path:
    - myapp.tf
    - foo
```

2.6 Module Configurations

2.6.1 CDK

Standard AWS CDK rules apply, with the following recommendations/caveats:

A `package.json` file is required, specifying the aws-cdk dependency. E.g.:

```
{
  "name": "mymodulename",
  "version": "1.0.0",
  "description": "My CDK module",
  "main": "index.js",
  "dependencies": {
    "@aws-cdk/cdk": "^0.9.2",
    "@types/node": "^10.10.1"
  },
  "devDependencies": {
    "aws-cdk": "^0.9.2",
    "typescript": "^3.0.3"
  }
  "author": "My Org",
  "license": "Apache-2.0"
}
```

We strongly recommend you commit the package-lock.json that is generated after running `npm install`

Build Steps

Build steps (e.g. for compiling TypeScript) can be specified in the module options. These steps will be run before each diff, deploy, or destroy.

```
deployments:
  - modules:
      - path: mycdkmodule
        environments:
          dev: true
        options:
          build_steps:
            - npx tsc
```

Environment Configs

Environments can be specified via deployment and/or module options. Each example below shows the explicit CDK `ACCOUNT/REGION` environment mapping; these can be alternately be specified with a simple boolean (e.g. `dev: true`).

Top-level Runway Config

```
---

deployments:
  - modules:
      - path: mycdkmodule
        environments:
          # CDK environment values can be specified in 3 forms:
          # Opt 1 - A yaml mapping, in which case each key:val pair will be provided␣
↪as context options
          # dev:
          #   route_53_zone_id: Z3P5QSUBK4POTI
          # Opt 2 - A string, in which case the explicit CDK ``ACCOUNT/REGION``␣
↪environment will be verified
          # dev: 987654321098/us-west-2
          # Opt 3 - Booleans, in which case the module will always be deployed in the␣
↪given environment
          # dev: true
```

and/or:

```
---

deployments:
  - environments:
      # CDK environment values can be specified in 3 forms:
      # Opt 1 - A yaml mapping, in which case each key:val pair will be provided as␣
↪context options
      # dev:
      #   route_53_zone_id: Z3P5QSUBK4POTI
      # Opt 2 - A string, in which case the explicit CDK ``ACCOUNT/REGION``␣
↪environment will be verified
      # dev: 987654321098/us-west-2
```

(continues on next page)

(continued from previous page)

```
      # Opt 3 - Booleans, in which case the module will always be deployed in the␣
↪given environment
      # dev: true
    modules:
      - mycdkmodule
```

In Module Directory

```
---
environments:
  # CDK environment values can be specified in 3 forms:
  # Opt 1 - A yaml mapping, in which case each key:val pair will be provided as␣
↪context options
  # dev:
  #   route_53_zone_id: Z3P5QSUBK4POTI
  # Opt 2 - A string, in which case the explicit CDK ``ACCOUNT/REGION`` environment␣
↪will be verified
  # dev: 987654321098/us-west-2
  # Opt 3 - Booleans, in which case the module will always be deployed in the given␣
↪environment
  # dev: true
```

(in `runway.module.yml`)

Disabling NPM CI

At the start of each module execution, Runway will execute `npm ci` to ensure the CDK is installed in the project (so Runway can execute it via `npx cdk`. This can be disabled (e.g. for use when the `node_modules` directory is pre-compiled) via the `skip_npm_ci` module option:

```
---
deployments:
  - modules:
      - path: mycdkproject.cdk
        options:
          skip_npm_ci: true
```

2.6.2 CloudFormation

CloudFormation modules are managed by 2 files:

- a key/value environment file
- a yaml file defining the stacks/templates/params.

Environment

Name these files in the form of `ENV-REGION.env` (e.g. `dev-us-east-1.env`) or `ENV.env` (e.g. `dev.env`):

```
# Namespace is used as each stack's prefix
# We recommend an (org/customer)/environment delineation
namespace: contoso-dev
environment: dev
customer: contoso
region: us-west-2
# The stacker bucket is the S3 bucket (automatically created) where templates
# are uploaded for deployment (a CloudFormation requirement for large templates)
stacker_bucket_name: stacker-contoso-us-west-2
```

Stack Config (yaml file)

These files can have any name ending in .yaml (they will be evaluated in alphabetical order):

```
# Note namespace/stacker_bucket_name being substituted from the environment
namespace: ${namespace}
stacker_bucket: ${stacker_bucket_name}

stacks:
  myvpcstack:  # will be deployed as contoso-dev-myvpcstack
    template_path: templates/vpc.yaml
    # The enabled option is optional and defaults to true. You can use it to
    # enable/disable stacks per-environment (i.e. like the namespace
    # substitution above, but with the value of either true or false for the
    # enabled option here)
    enabled: true
  myvpcendpoint:
    template_path: templates/vpcendpoint.yaml
    # variables map directly to CFN parameters; here used to supply the
    # VpcId output from the myvpcstack to the VpcId parameter of this stack
    variables:
      VpcId: ${output myvpcstack::VpcId}
```

The config yaml supports many more features; see the full CFNgin documentation for more detail (e.g. stack configuration options, additional lookups in addition to output (e.g. SSM, DynamoDB))

Environment Values Via Runway Deployment/Module Options

In addition or in place of the environment file(s), deploy environment specific values can be provided via deployment and module options as `parameters`. It is recommended to use Lookups in the `parameters` section to assist in selecting the appropriate values for the deploy environment and/or region being deployed to but, this is not a requirement if the value will remain the same.

Top-level Runway Config

```
---

deployments:
  - modules:
      - path: mycfnstacks
        parameters:
          namespace: contoso-${env DEPLOY_ENVIRONMENT}
          foo: bar
          some_value: ${var some_map.${env DEPLOY_ENVIRONMENT}}
```

and/or

```
---

deployments:
  - parameters:
      namespace: contoso-${env DEPLOY_ENVIRONMENT}
      foo: bar
      some_value: ${var some_map.${env DEPLOY_ENVIRONMENT}}
    modules:
      - mycfnstacks
```

In Module Directory

```
---
parameters:
  namespace: contoso-dev
  foo: bar
```

(in `runway.module.yml`)

2.6.3 Custom Plugin Support

Need to expand Runway to wrap other tools? Yes - you can do that with custom plugin support.

Overview

Runway can import Python modules that can perform custom deployments with your own set of Runway modules. Let's say for example you want to have Runway execute an Ansible playbook to create an EC2 security group as one of the steps in the middle of your Runway deployment list - this is possible with your own plugin. The custom plugin support allows you to mix-and-match natively supported modules (e.g. CloudFormation, Terraform) with plugins you write providing additional support for non-native modules. Although written in Python, these plugins can natively execute non Python binaries.

RunwayModule Class

Runway provides a Python Class named `RunwayModule` that can be imported into your custom plugin/Python module. This base class will give you the ability to write your own module that can be added to your runway.yml deployment list (More info on runway.yml below). There are three required functions:

```
- plan - This code block gets called when ``runway taxi`` executes
- deploy - This code block gets called when ``runway takeoff`` executes
- destroy - This code block gets called when ``runway destroy`` executes
```

All of these functions are required, but are permitted to be empty no-op/pass statements if applicable.

Context Object

`self.context` includes many helpful resources for use in your Python module. Some notable examples are:

```
- self.context.env_name - name of the environment
- self.context.env_region - region in which the module is being executed
- self.context.env_vars - OS environment variables provided to the module
- self.path - path to your Runway module folder
```

runway.yml Example

After you have written your plugin, you need to add the module `class_path` to your module's configuration. Below is an example `runway.yml` containing a single module that looks for an Ansible playbook in a folder at the root of your Runway environment (i.e. repo) named "security_group.ansible".

Setting `class_path` tells Runway to import the DeployToAWS Python class, from a file named Ansible.py in a folder named "local_runway_extensions" (Standard Python import conventions apply). Runway will execute the `deploy` function in your class when you perform a `runway deploy` (AKA takeoff).

```
deployments:
  - modules:
      - path: security_group.ansible
        class_path: local_runway_extensions.Ansible.DeployToAWS
    regions:
      - us-east-1
```

Below is the `Ansible.py` module referenced above that wraps the `ansible-playbook` command. It will be responsible for deploying an EC2 Security Group from the playbook with a naming convention of `<env>-<region>.yaml` within a fictional `security_group.ansible` Runway module folder. In this example, the `ansible-playbook` binary would already have been installed prior to a Runway deploy, but this example does check to see if it is installed before execution and logs an error if not. The Runway plugin will only execute the ansible-playbook against a `yaml` file associated with the environment and set for the Runway execution and region defined in the `runway.yml`.

Using the above `runway.yml` and the plugin/playbook below saved to the Runway module folder you will only have a deployment occur in the `dev` environment in `us-east-1`. If you decide to perform a runway deployment in the `prod` environment, or in a different region, the ansible-playbook deployment will be skipped. This matches the behavior of the Runway's native modules.

```
"""Ansible Plugin example for Runway."""

import logging
```

(continues on next page)

(continued from previous page)

```
import subprocess
import sys
import os

from runway.module import RunwayModule
from runway.util import which

LOGGER = logging.getLogger('runway')


def check_for_playbook(playbook_path):
    """Determine if environment/region playbook exists."""
    if os.path.isfile(playbook_path):
        LOGGER.info("Processing playbook: %s", playbook_path)
        return {'skipped_configs': False}
    else:
        LOGGER.error("No playbook for this environment/region found -- "
                     "looking for %s", playbook_path)
        return {'skipped_configs': True}


class DeployToAWS(RunwayModule):
    """Ansible Runway Module."""

    def plan(self):
        """Skip plan"""
        LOGGER.info('plan not currently supported for Ansible')
        pass

    def deploy(self):
        """Run ansible-playbook."""
        if not which('ansible-playbook'):
            LOGGER.error('"ansible-playbook" not found in path or is not '
                         'executable; please ensure it is installed'
                         'correctly.')
            sys.exit(1)
        playbook_path = (self.path + "-" + self.context.env_name + self.context.env_
↪region)
        response = check_for_playbook(playbook_path)
        if response['skipped_configs']:
            return response
        else:
            subprocess.check_output(
                ['ansible-playbook', playbook_path])
            return response

    def destroy(self):
        """Skip destroy."""
        LOGGER.info('Destroy not currently supported for Ansible')
        pass
```

And below is the example Ansible playbook itself, saved as `dev-us-east-1.yaml` in the security_group.ansible folder:

```
- hosts: localhost
  connection: local
```

(continues on next page)

(continued from previous page)

```
  gather_facts: false
  tasks:
      - name: create a security group in us-east-1
        ec2_group:
          name: dmz
          description: Dev example ec2 group
          region: us-east-1
          rules:
            - proto: tcp
              from_port: 80
              to_port: 80
              cidr_ip: 0.0.0.0/0
        register: security_group
```

The above would be deployed if `runway deploy` was executed in the `dev` environment to `us-east-1`.

2.6.4 Kubernetes

Kubernetes manifests can be deployed via Runway, offering an ideal way to handle core infrastructure-layer (e.g. shared ConfigMaps & Service Accounts) configuration of clusters. Perform the following steps to align your k8s directories with Runway's requirements & best practices.

Part 1: Adding Kubernetes to Deployment

Start by adding your Kustomize overlay organized Kubernetes directory to your runway.yml's list of modules.

Directory tree:

```
.
├── runway.yml
└── kubernetesstuff.k8s
    ├── base
    │   ├── kustomization.yaml
    │   └── service.yaml
    └── overlays
        ├── prod
        │   └── kustomization.yaml
        └── staging
            └── kustomization.yaml
```

runway.yml:

```
---
deployments:
  - modules:
      - kubernetesstuff.k8s
    regions:
      - us-east-1
```

Each overlay's kustomization can be as simple as including the base directory and (optionally) adding a resource prefix. E.g., in the staging directory's kustomize.yml:

```
bases:
  - ../base
namePrefix: staging-
```

The base directory's kustomization then in turn includes the base directory's manifests:

```
resources:
  - service.yaml
```

Part 2: Specify the Kubectl Version

By specifying the version via a `.kubectl-version` file in your overlay directory, or a module option, Runway will automatically download & use that version for the module. This is recommended to keep a predictable experience when deploying your module.

.kubectl-version:

```
1.14.5
```

or in runway.yml, either for a single module:

```
---
deployments:
  - modules:
      - path: myk8smodule
        options:
          kubectl_version:
            "*": 1.14.5  # applies to all environments
            # prod: 1.13.0  # can also be specified for a specific environment
```

and/or for a group of modules:

```
---
deployments:
  - modules:
      - path: myk8smodule
      - path: anotherk8smodule
    module_options:  # shared between all modules in deployment
      kubectl_version:
        "*": 1.14.5  # applies to all environments
        # prod: 1.13.0  # can also be specified for a specific environment
```

Without a version specified, Runway will fallback to whatever `kubectl` it finds first in your PATH.

Part 3: Setting KUBECONFIG location

If using a non-default kubeconfig location, you can provide it using Runway's option for setting environment variables. This can be set as a relative path or an absolute one. E.g.:

```
---
deployments:
  - modules:
      - path: myk8smodule
        options:
          kubectl_version:
  - regions:
      - us-east-1
env_vars:
  staging:
```

(continues on next page)

(continued from previous page)

```
    KUBECONFIG:
      - .kube
      - staging
      - config
  prod:
    KUBECONFIG:
      - .kube
      - prod
      - config
```

(this would set `KUBECONFIG` to `<path_to_runway.yml>/.kube/staging/config` in the staging environment)

2.6.5 Serverless

Standard Serverless rules apply, with the following recommendations/caveats:

- Runway environments map directly to Serverless stages.
- A `package.json` file is required, specifying the serverless dependency, e.g.:

```
{
  "name": "mymodulename",
  "version": "1.0.0",
  "description": "My serverless module",
  "main": "handler.py",
  "devDependencies": {
    "serverless": "^1.25.0"
  },
  "author": "Serverless Devs",
  "license": "ISC"
}
```

- We strongly recommend you commit the package-lock.json that is generated after running `npm install`
- Each stage requires either its own variables file (even if empty for a particular stage) in one of the following forms, or a configured environment in the module options (see `Enabling Environments Via Runway Deployment/Module Options` below):
- `env/STAGE-REGION.yml`
- `config-STAGE-REGION.yml`
- `env/STAGE.yml`
- `config-STAGE.yml`
- `env/STAGE-REGION.json`
- `config-STAGE-REGION.json`
- `env/STAGE.json`
- `config-STAGE.json`

Enabling Environments Via Runway Deployment/Module Options

Environments can be specified via deployment and module options in lieu of variable files.

Top-level Runway Config

```
---

deployments:
  - modules:
      - path: myslsmodule
        environments:
          dev: true
          prod: true
```

and/or

```
---

deployments:
  - environments:
      dev: true
      prod: true
    modules:
      - myslsmodule
```

In Module Directory

```
---
environments:
  dev: true
  prod: true
```

(in `runway.module.yml`)

Promoting Builds Through Environments

Serverless build `.zips` can be used between environments by setting the `promotezip` module option and providing a bucket name in which to cache the builds.

The first time the Serverless module is deployed using this option, it will build/deploy as normal and cache the artifact on S3. On subsequent deploys, Runway will used that cached artifact (finding it by comparing the module source code).

This enables a common build account to deploy new builds in a dev/test environment, and then promote that same zip through other environments (any of these environments can be in the same or different AWS accounts).

The CloudFormation stack deploying the zip will be re-generated on each deployment (so environment-specific values/lookups will work as normal).

Example config:

```
---
deployments:
  - modules:
    - path: myslsproject.sls
      options:
        promotezip:
          bucketname: my-build-account-bucket-name
```

Disabling NPM CI

At the start of each module execution, Runway will execute `npm ci` to ensure Serverless Framework is installed in the project (so Runway can execute it via `npx sls`. This can be disabled (e.g. for use when the `node_modules` directory is pre-compiled) via the `skip_npm_ci` module option:

```
---
deployments:
  - modules:
      - path: myslsproject.sls
        options:
          skip_npm_ci: true
```

Specifying Serverless CLI Arguments/Options

Runway can pass custom arguments/options to the Serverless CLI by using the `args` option. These will always be placed after the default arguments/options

The value of `args` must be a list of arguments/options to pass to the CLI. Each element of the argument/option should be it's own list item (e.b. `--config sls.yml` would be `['--config', 'sls.yml']`.

Important: Do not provide `--region <region>` or `--stage <stage>` here. These will be provided by Runway.

Runway Example

```
---
deployments:
  - modules:
      - path: sampleapp.sls
        options:
          args:
            - '--config'
            - sls.yml
    regions
      - us-east-2
    environments:
      example: true
```

Command Equivalent

```
serverless deploy -r us-east-1 --stage example --config sls.yml
```

2.6.6 Static Site

This module type performs idempotent deployments of static websites. It combines CloudFormation stacks (for S3 buckets & CloudFront Distribution) with additional logic to build & sync the sites.

It can be used with a configuration like the following:

```
deployments:
  - modules:
      - path: web
        class_path: runway.module.staticsite.StaticSite
        parameters:
          namespace: contoso-dev
          staticsite_aliases: web.example.com,foo.web.example.com
          staticsite_acmcert_arn: arn:aws:acm:us-east-1:123456789012:certificate/...
        options:
          build_steps:
            - npm ci
            - npm run build
          build_output: dist
    regions:
      - us-west-2
```

This will build the website in `web` via the specified build_steps and then upload the contents of `web/dist` to an S3 bucket created in the CloudFormation stack `web-dev-conduit`. On subsequent deploys, the website will be built and synced only if the non-git-ignored files in `web` change.

The site domain name is available via the `CFDistributionDomainName` output of the `<namespace>-<path>` stack (e.g. `contoso-dev-web` above) and will be displayed on stack creation/updates.

A start-to-finish example walkthrough is available in the *Conduit quickstart*.

Please note: The CloudFront distribution will take a significant amount of time to spin up on initial deploy (10 to 60 minutes is not abnormal). Incorporating CloudFront with a static site is a common best practice, however, if you are working on a development project it may benefit you to add the *staticsite_cf_disable* environment parameter set to *true*.

Example of all Static Site Options

Most of these options are not required, but are listed here for reference:

```
deployments:
  - modules:
      - name: conduitsite  # defaults to path; used in stack names & ssm parameter
        path: web
        class_path: runway.module.staticsite.StaticSite
        parameters:
          # The only required parameter value is namespace
          namespace: contoso-${env DEPLOY_ENVIRONMENT}
          staticsite_acmcert_arn: arn:aws:acm:us-east-1:123456789012:certificate/...
```

(continues on next page)

(continued from previous page)

```
          # A cert ARN can also be looked up dynamically via SSM
          staticsite_acmcert_ssm_param: us-west-2@MySSMParamName...

          staticsite_aliases: example.com,foo.example.com
          staticsite_web_acl: arn:aws:waf::123456789012:webacl/...

          # staticsite_enable_cf_logging defaults to true
          staticsite_enable_cf_logging: true

          # Deploy Lambda@Edge to rewrite directory indexes
          # e.g. support accessing example.org/foo/
          staticsite_rewrite_directory_index: index.html

          # You can also deploy custom Lambda@Edge associations with your
          # pre-built function versions
          # (this takes precedence over staticsite_rewrite_directory_index)
          staticsite_lambda_function_associations:
            - type: origin-request
              arn: arn:aws:lambda:us-east-1:123456789012:function:foo:1

          # Custom error response options can be defined
          staticsite_custom_error_responses:
            - ErrorCode: 404
              ResponseCode: 200
              ResponsePagePath: /index.html

          # Don't use CloudFront with the site
          # i.e. for a development site accessible only via its S3-url
          statisite_cf_disable: true
        options:
          pre_build_steps:  # commands to run before generating hash of files
            - command: npm ci
              cwd: ../myothermodule  # directory relative to top-level path setting
            - command: npm run export
              cwd: ../myothermodule
          source_hashing:  # overrides for source hash collection/tracking
            enabled: true  # if false, build & upload will occur on every deploy
            parameter: /${namespace}/myparam  # defaults to <namespace>-<name/path>-
↪hash
            directories:  # overrides default hash directory of top-level path setting
              - path: ./
              - path: ../common
                # Additional (gitignore-format) exclusions to hashing
                # (.gitignore files are loaded automatically)
                exclusions:
                  - foo/*
          build_steps:
            - npm ci
            - npm run build
          build_output: dist  # overrides default directory of top-level path setting
    regions:
      - us-west-2
```

2.6.7 Terraform

Runway provides a simple way to run the Terraform versions you want with variable values specific to each environment. Perform the following steps to align your Terraform directory with Runway's requirements & best practices.

Part 1: Adding Terraform to Deployment

Start by adding your Terraform directory to your runway.yml's list of modules.

(Note: to Runway, a module is just a directory in which to run `terraform apply`, `serverless deploy`, etc - no relation to Terraform's concept of modules)

Directory tree:

```
.
├── runway.yml
└── terraformstuff.tf
    └── main.tf
```

runway.yml:

```
---
deployments:
  - modules:
      - terraformstuff.tf
    regions:
      - us-east-1
```

Part 2: Specify the Terraform Version

By specifying the version via a `.terraform-version` file in your Terraform directory, or a module option, Runway will automatically download & use that version for the module. This, alongside tightly pinning Terraform provider versions, is highly recommended to keep a predictable experience when deploying your module.

.terraform-version:

```
0.11.6
```

or in runway.yml, either for a single module:

```
---
deployments:
  - modules:
      - path: mytfmodule
        options:
          terraform_version:
            "*": 0.11.13  # applies to all environments
            # prod: 0.9.0  # can also be specified for a specific environment
```

and/or for a group of modules:

```
---
deployments:
  - modules:
      - path: mytfmodule
      - path: anothermytfmodule
```

(continues on next page)

(continued from previous page)

```
    module_options:  # shared between all modules in deployment
      terraform_version:
        "*": 0.11.13  # applies to all environments
        # prod: 0.9.0  # can also be specified for a specific environment
```

Without a version specified, Runway will fallback to whatever `terraform` it finds first in your PATH.

Part 3: Adding Backend Configuration

Next, configure the backend for your Terraform configuration. If your Terraform will only ever be used with a single backend, it can be defined inline:

main.tf:

```
terraform {
  backend "s3" {
    region = "us-east-1"
    key = "some_unique_identifier_for_my_module" # e.g. contosovpc
    bucket = "some_s3_bucket_name"
    dynamodb_table = "some_ddb_table_name"
  }
}
```

However, it's generally preferable to separate the backend configuration out from the rest of the Terraform code. Choose from one of the following options.

Backend Config in File

Backend config options can be specified in a separate file or multiple files per environment and/or region:

- `backend-ENV-REGION.tfvars`
- `backend-ENV.tfvars`
- `backend-REGION.tfvars`
- `backend.tfvars`

```
region = "us-east-1"
bucket = "some_s3_bucket_name"
dynamodb_table = "some_ddb_table_name"
```

In the above example, where all but the key are defined, the main.tf backend definition is reduced to:

main.tf:

```
terraform {
  backend "s3" {
    key = "some_unique_identifier_for_my_module" # e.g. contosovpc
  }
}
```

Backend Config in runway.yml

Backend config options can also be specified as a module option in runway.yml:

Either for a single module:

```
---
deployments:
  - modules:
      - path: mytfmodule
        options:
          terraform_backend_config:
            bucket: mybucket
            region: us-east-1
            dynamodb_table: mytable
```

and/or for a group of modules:

```
---
deployments:
  - modules:
      - path: mytfmodule
      - path: anothermytfmodule
    module_options:  # shared between all modules in deployment
      terraform_backend_config:
        bucket: mybucket
        region: us-east-1
        dynamodb_table: mytable
```

Backend CloudFormation Outputs in runway.yml

A recommended option for managing the state bucket and table is to create them via CloudFormation (try running `runway gen-sample cfn` to get a template and stack definition for bucket/table stack). To further support this, backend config options can be looked up directly from CloudFormation outputs.

Either for a single module:

```
---
deployments:
  - modules:
      - path: mytfmodule
        options:
          terraform_backend_config:
            region: us-east-1
          terraform_backend_cfn_outputs:
            bucket: StackName::OutputName  # e.g. common-tf-
↪state::TerraformStateBucketName
            dynamodb_table: StackName::OutputName  # e.g. common-tf-
↪state::TerraformStateTableName
```

and/or for a group of modules:

```
---
deployments:
  - modules:
      - path: mytfmodule
```

(continues on next page)

(continued from previous page)

```
      - path: anothermytfmodule
    module_options:  # shared between all modules in deployment
      terraform_backend_config:
        region: us-east-1
      terraform_backend_cfn_outputs:
        bucket: StackName::OutputName  # e.g. common-tf-
↪state::TerraformStateBucketName
        dynamodb_table: StackName::OutputName  # e.g. common-tf-
↪state::TerraformLockTableName
```

Backend SSM Parameters in runway.yml

Similar to the CloudFormation lookup, backend config options can be looked up directly from SSM Parameters.

Either for a single module:

```
---
deployments:
  - modules:
      - path: mytfmodule
        options:
          terraform_backend_config:
            region: us-east-1
          terraform_backend_ssm_params:
            bucket: ParamNameHere
            dynamodb_table: ParamNameHere
```

and/or for a group of modules:

```
---
deployments:
  - modules:
      - path: mytfmodule
      - path: anothermytfmodule
    module_options:  # shared between all modules in deployment
      terraform_backend_config:
        region: us-east-1
      terraform_backend_ssm_params:
        bucket: ParamNameHere
        dynamodb_table: ParamNameHere
```

Part 4: Variable Values

Finally, define your per-environment variables using one or both of the following options.

Values in Variable Definitions Files

Standard Terraform tfvars files can be used, exactly as one normally would with `terraform apply -var-file`. Runway will automatically detect them when named like `ENV-REGION.tfvars` or `ENV.tfvars`.

E.g. `common-us-east-1.tfvars`:

```
region = "us-east-1"
image_id = "ami-abc123"
```

Values in runway.yml

Variable values can also be specified as parameter values in runway.yml. It is recommended to use Lookups in the `parameters` section to assist in selecting the appropriate values for the deploy environment and/or region being deployed to but, this is not a requirement if the value will remain the same.

```
---

deployments:
  - modules:
      - path: mytfmodule
        parameters:
          region: ${env AWS_REGION}
          image_id: ${var image_id.${env AWS_REGION}}
          mylist:
            - item1
            - item2
          mymap:
            key1: value1
            key2: value1
```

and/or

```
---

deployments:
  - parameters:
      region: ${env AWS_REGION}
      image_id: ${var image_id.${env AWS_REGION}}
      mylist:
        - item1
        - item2
      mymap:
        key1: value1
        key2: value1
    modules:
      - mytfmodule
```

2.7 CFNgin

CFNgin is a library (originating from the open source library Stacker) used to create & update CloudFormation stacks.

It provides a simple way to manage stacks with features like:

- Automatic stack ordering (e.g. deploy stack "A" before stacks "B" & "C")
- Per-environment values for stack parameters
- Actions before & after stack creation/deletion

Contents:

2.7.1 Migrating from Stacker to CFNgin

Important: Most current uses of Runway with Stacker will continue to work. But, for imports from Stacker, Runway will automatically redirect them to CFNgin. Because of this, you may experience errors depending on how you are consuming the Stacker components. This "shim" will remain in place until the release of Runway 2.0.0, no sooner then 2020-12.

Blueprints

All components available in Stacker 1.7.0 are available in CFNgin at the same path within `runway.cfngin`.

Example

```
# what use to be this
from stacker.blueprints.base import Blueprint
from stacker.blueprints.variables.types import CFNString

# now becomes this
from runway.cfngin.blueprints.base import Blueprint
from runway.cfngin.blueprints.variables.types import CFNString
```

Config Files

There are some config top-level keys that have changed when used CFNgin. Below is a table of the Stacker key and what they have been changed to for CFNgin

Important: The Stacker keys can still be used with CFNgin for the time being. This will remain in place until the release of Runway 2.0.0, no sooner then 2020-12.

Stacker	CFNgin
`stacker_bucket`	`cfngin_bucket`
`stacker_bucket_region`	`cfngin_bucket_region`
`stacker_cache_dir`	`cfngin_cache_dir`

Build-in Hooks

All hooks available in Stacker 1.7.0 are available in CFNgin at the same path within `runway.cfngin`.

Example Definition

```
pre_build:
  what_use_to_be_this:
    path: stacker.hooks.commands.run_command
    args:
      command: echo "Hello $USER!"
  now_becomes_this:
    path: runway.cfngin.hooks.commands.run_command
    args:
      command: echo "Hello $USER!"
```

See also:

CFNgin API Docs

Custom Lookups

See the Custom Lookups section of the docs for detailed instructions on how lookups should be written.

Important: Stacker lookups (function and class styles) are supported for the time being. It is recommended to update them to the CFNgin format outlined in Custom Lookups. Support for Stacker style lookups will remain in place until the release of Runway 2.0.0, no sooner then 2020-12.

2.7.2 CFNgin Config File

CFNgin makes use of a YAML formatted config file to define the different CloudFormation stacks that make up a given environment.

The configuration file has a loose definition, with only a few top-level keywords. Other than those keywords, you can define your own top-level keys to make use of other YAML features like anchors & references to avoid duplicating config. (See *YAML anchors & references* for details)

Top Level Keywords

Namespace

You can provide a `namespace` to create all stacks within. The *namespace* will be used as a prefix for the name of any stack that CFNgin creates.

In addition, this value will be used to create an S3 bucket that CFNgin will use to upload and store all CloudFormation templates.

In general, this is paired with the concept of Environments to create a *namespace* per environment.

```
namespace: ${namespace}
```

Namespace Delimiter

By default, CFNgin will use - as a delimiter between your *namespace* and the declared stack name to build the actual CloudFormation stack name that gets created. Since child resources of your stacks will, by default, use a portion of your stack name in the auto-generated resource names, the first characters of your fully-qualified stack name potentially convey valuable information to someone glancing at resource names. If you prefer to not use a delimiter, you can pass the `namespace_delimiter` top-level keyword in the config as an empty string.

See the CloudFormation API Reference for allowed stack name characters

S3 Bucket

CFNgin, by default, pushes your CloudFormation templates into an S3 bucket and points CloudFormation at the template in that bucket when launching or updating your stacks. By default it uses a bucket named `stacker-${namespace}`, where the *namespace* is the *namespace* provided the config.

If you want to change this, provide the `cfngin_bucket` top-level keyword in the config.

The bucket will be created in the same region that the stacks will be launched in. If you want to change this, or if you already have an existing bucket in a different region, you can set the `cfngin_bucket_region` to the region where you want to create the bucket.

If you want CFNgin to upload templates directly to CloudFormation, instead of first uploading to S3, you can set `cfngin_bucket` to an empty string. However, note that template size is greatly limited when uploading directly. See the CloudFormation Limits Reference.

Persistent Graph

Each time CFNgin is run, it creates a dependency graph of Stacks. This is used to determine the order in which to execute them. This graph can be persisted between runs to track the removal of Stacks the config file.

When a stack is present in the persistent graph but not in the graph constructed from the config file, CFNgin will delete the stack from CloudFormation. This takes effect during both build and destroy actions.

The persistent graph is also used with the graph command where it is merged with the graph constructed from the config file.

To enable persistent graph, set **persistent_graph_key** to a unique value that will be used to construct the path to the persistent graph object in S3. This object is stored in the CFNgin *S3 Bucket* which is also used for CloudFormation templates. The fully qualified path to the object will look like the below.

```
s3://${cfngin_bucket}/${namespace}/persistent_graphs/${namespace}/${persistent_graph_
↪key}.json
```

Note: It is recommended to enable versioning on the CFNgin *S3 Bucket* when using persistent graph to have a backup version in the event something unintended happens. A warning will be logged if this is not enabled.

If CFNgin creates an *S3 Bucket* for you when persistent graph is enabled, it will be created with versioning enabled.

Important: When choosing a value for **persistent_graph_key**, it is vital to ensure the value is unique for the **namespace** being used. If the key is a duplicate, stacks that are not intended to be destroyed will be destroyed.

When executing an action that will be modifying the persistent graph (build or destroy), the S3 object is *"locked"*. The lock is a tag applied to the object at the start of one of these actions. The tag-key is **cfngin_lock_code** and the tag-value is UUID generated each time a command is run. In order for CFNgin to lock a persistent graph object, the tag must not be present on the object. For CFNgin to act on the graph (modify or unlock) the value of the tag must match the UUID of the current CFNgin session. If the object is locked or the code does not match, an error will be raised and no action will be taken. This prevents two parties from acting on the same persistent graph object concurrently which would create a race condition.

Note: A persistent graph object can be unlocked manually by removing the **cfngin_lock_code** tag from it. This should be done with caution as it will cause any active sessions to raise an error.

Persistent Graph Example

config.yml

```yaml
namespace: example
cfngin_bucket: cfngin-bucket
persistent_graph_key: my_graph  # .json - will be appended if not provided
stacks:
  first_stack:
    ...
  new_stack:
    ...
```

s3://cfngin-bucket/persistent_graphs/example/my_graph.json

```
{
  "first_stack": [],
  "removed_stack": [
    "first_stack"
  ]
}
```

Result

Given the above config file and persistent graph, when running `runway deploy`, the following will occur.

1. The `{"Key": "cfngin_lock_code", "Value": "123456"}` tag is applied to **s3://cfngin-bucket/persistent_graphs/example/my_graph.json** to lock it to the current session.
2. **removed_stack** is deleted from CloudFormation and deleted from the persistent graph object in S3.
3. **first_stack** is updated in CloudFormation and updated in the persistent graph object in S3 (incase dependencies change).
4. **new_stack** is created in CloudFormation and added to the persistent graph object in S3.
5. The `{"Key": "cfngin_lock_code", "Value": "123456"}` tag is removed from **s3://cfngin-bucket/persistent_graphs/example/my_graph.json** to unlock it for use in other sessions.

Module Paths

When setting the `classpath` for Blueprints/hooks, it is sometimes desirable to load modules from outside the default `sys.path` (e.g., to include modules inside the same repo as config files).

Adding a path (e.g. `./`) to the `sys_path` top-level keyword will allow modules from that path location to be used.

Service Role

By default CFNgin doesn't specify a service role when executing changes to CloudFormation stacks. If you would prefer that it do so, you can set `service_role` to be the ARN of the role that CFNgin should use when executing CloudFormation changes.

This is the equivalent of setting `RoleARN` on a call to the following CloudFormation api calls: `CreateStack`, `UpdateStack`, `CreateChangeSet`.

See the AWS documentation for AWS CloudFormation Service Roles.

Remote Packages

The `package_sources` top-level keyword can be used to define remote sources for Blueprints (e.g., retrieving `src/runway/blueprints` on github at tag `v1.3.7`).

The only required key for a git repository config is `uri`, but `branch`, `tag`, & `commit` can also be specified.

```
package_sources:
  git:
    - uri: git@github.com:onicagroup/runway.git
    - uri: git@github.com:onicagroup/runway.git
      tag: 1.0.0
      paths:
        - src/runway/blueprints
    - uri: git@github.com:contoso/webapp.git
      branch: staging
    - uri: git@github.com:contoso/foo.git
      commit: 12345678
```

If no specific commit or tag is specified for a repo, the remote repository will be checked for newer commits on every execution of CFNgin.

For `.tar.gz` & `zip` archives on s3, specify a `bucket` & `key`.

```
package_sources:
  s3:
    - bucket: mycfngins3bucket
      key: archives/blueprints-v1.zip
      paths:
        - blueprints
    - bucket: anothers3bucket
      key: public/public-blueprints-v2.tar.gz
      requester_pays: true
    - bucket: yetanothers3bucket
      key: sallys-blueprints-v1.tar.gz
      # use_latest defaults to true - will update local copy if the
      # last modified date on S3 changes
      use_latest: false
```

Local directories can also be specified.

```
package_sources:
  local:
    - source: ../vpc
```

Use the `paths` option when subdirectories of the repo/archive/directory should be added to CFNgins's `sys.path`.

Cloned repos/archives will be cached between builds; the cache location defaults to `~/.runway_cache` but can be manually specified via the `cfngin_cache_dir` top-level keyword.

Remote Configs

Configuration YAMLs from remote configs can also be used by specifying a list of `configs` in the repo to use.

```
package_sources:
  git:
    - uri: git@github.com:acmecorp/cfngin_blueprints.git
      configs:
        - vpc.yaml
```

In this example, the configuration in `vpc.yaml` will be merged into the running current configuration, with the current configuration's values taking priority over the values in `vpc.yaml`.

Dictionary Stack Names & Hook Paths

To allow remote configs to be selectively overridden, stack names & hook paths are defined as dictionaries.

```
pre_build:
  my_route53_hook:
    path: runway.cfngin.hooks.route53.create_domain:
    required: true
    enabled: true
    args:
      domain: mydomain.com
stacks:
  vpc-example:
    class_path: cfngin_blueprints.vpc.VPC
    locked: false
    enabled: true
  bastion-example:
    class_path: cfngin_blueprints.bastion.Bastion
    locked: false
    enabled: true
```

Pre & Post Hooks

Many actions allow for pre & post hooks. These are python functions/methods that are executed before, and after the action is taken for the entire config. Hooks can be enabled or disabled, per hook. Only the following actions allow pre/post hooks:

- build (keywords: `pre_build`, `post_build`)
- destroy (keywords: `pre_destroy`, `post_destroy`)

There are a few reasons to use these, though the most common is if you want better control over the naming of a resource than what CloudFormation allows.

The keyword is a dictionary with the following keys:

path: the python import path to the hook.

data_key: If set, and the hook returns data (a dictionary), the results will be stored in the `context.hook_data` with the `data_key` as its key.

required: Whether to stop execution if the hook fails.

enabled: Whether to execute the hook every CFNgin run. Default: True. This is a bool that grants you the ability to execute a hook per environment when combined with a variable pulled from an environment file.

args: A dictionary of arguments to pass to the hook with support for lookups. Note that lookups that change the order of execution, like `output`, can only be used in a *post* hook but hooks like `rxref` are able to be used with either *pre* or *post* hooks.

An example using the `create_domain` hook for creating a route53 domain before the build action:

```
pre_build:
  create_my_domain:
    path: runway.cfngin.hooks.route53.create_domain
    required: true
    enabled: true
    args:
      domain: mydomain.com
```

An example of a hook using the `create_domain_bool` variable from the environment file to determine if the hook should run. Set `create_domain_bool: true` or `create_domain_bool: false` in the environment file to determine if the hook should run in the environment CFNgin is running against:

```
pre_build:
  create_my_domain:
    path: runway.cfngin.hooks.route53.create_domain
    required: true
    enabled: ${create_domain_bool}
    args:
      domain: mydomain.com
```

An example of a custom hooks using various lookups in it's arguments:

```
pre_build:
  custom_hook1:
    path: path.to.hook1.entry_point
    args:
      ami: ${ami [<region>@]owners:self,888888888888,amazon name_regex:server[0-9]+␣
↪architecture:i386}
      user_data: ${file parameterized-64:file://some/path}
```

(continues on next page)

(continued from previous page)

```
      db_endpoint: ${rxref some-stack::Endpoint}
      subnet: ${xref some-stack::Subnet}
      db_creds: ${ssmstore us-east-1@MyDBUser}
  custom_hook2:
    path: path.to.hook.entry_point
    args:
      bucket: ${dynamodb us-east-1:TestTable@TestKey:TestVal.BucketName}
      bucket_region: ${envvar AWS_REGION}  # this variable is set by Runway
      files:
        - ${file plain:file://some/path}

post_build:
  custom_hook3:
    path: path.to.hook3.entry_point
    args:
      nlb: ${output nlb-stack::Nlb}  # output can only be used as a post hook
```

Tags

CloudFormation supports arbitrary key-value pair tags. All stack-level, including automatically created tags, are propagated to resources that AWS CloudFormation supports. See AWS CloudFormation Resource Tags Type for more details. If no tags are specified, the `cfngin_namespace` tag is applied to your stack with the value of `namespace` as the tag value.

If you prefer to apply a custom set of tags, specify the top-level keyword `tags` as a map.

Example:

```
tags:
  "hello": world
  "my_tag:with_colons_in_key": ${dynamic_tag_value_from_my_env}
  simple_tag: simple value
```

If you prefer to have no tags applied to your stacks (versus the default tags that CFNgin applies), specify an empty map for the top-level keyword.

```
tags: {}
```

Mappings

Mappings are dictionaries that are provided as Mappings to each CloudFormation stack that CFNgin produces.

These can be useful for providing things like different AMIs for different instance types in different regions.

```
mappings:
  AmiMap:
    us-east-1:
      NAT: ami-ad227cc4
      ubuntu1404: ami-74e27e1c
      bastion: ami-74e27e1c
    us-west-2:
      NAT: ami-290f4119
```

(continues on next page)

(continued from previous page)

```
    ubuntu1404: ami-5189a661
    bastion: ami-5189a661
```

These can be used in each Blueprint/stack as usual.

Lookups

Lookups allow you to create custom methods which take a value and are resolved at build time. The resolved values are passed to the Blueprint before it is rendered. For more information, see the Lookups documentation.

CFNgin provides some common lookups, but it is sometimes useful to have your own custom lookup that doesn't get shipped with Runway. You can register your own lookups by defining a `lookups` key.

```
lookups:
  custom: path.to.lookup.handler
```

The key name for the lookup will be used as the type name when registering the lookup. The value should be the path to a valid lookup handler.

You can then use these within your config.

```
conf_value: ${custom some-input-here}
```

Stacks

This is the core part of the config - this is where you define each of the stacks that will be deployed in the environment. The top-level keyword `stacks` is populated with a dictionary, each representing a single stack to be built.

They key used in the dictionary of stacks is used as the logical name of the stack. The value here must be unique within the config. If no `stack_name` is provided, the value here will be used for the name of the CloudFormation stack.

A stack has the following keys:

class_path: The python class path to the Blueprint to be used. Specify this or `template_path` for the stack.

template_path: Path to raw CloudFormation template (JSON or YAML). Specify this or `class_path` for the stack. Path can be specified relative to the current working directory (e.g. templates stored alongside the Config), or relative to a directory in the python `sys.path` (i.e. for loading templates retrieved via `packages_sources`).

description: A short description to apply to the stack. This overwrites any description provided in the Blueprint. See: http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-description-structure.html

variables: A dictionary of *Variables* to pass into the Blueprint when rendering the CloudFormation template. *Variables* can be any valid YAML data structure.

locked: (optional) If set to true, the stack is locked and will not be updated unless the stack is passed to CFNgin via the `--force` flag. This is useful for **risky** stacks that you don't want to take the risk of allowing CloudFormation to update, but still want to make sure get launched when the environment is first created. When `locked`, it's not necessary to specify a `class_path` or `template_path`.

enabled: (optional) If set to false, the stack is disabled, and will not be built or updated. This can allow you to disable stacks in different environments.

protected: (optional) When running an update in non-interactive mode, if a stack has `protected` set to `true` and would get changed, CFNgin will switch to interactive mode for that stack, allowing you to approve/skip the change.

requires: (optional) a list of other stacks this stack requires. This is for explicit dependencies - you do not need to set this if you refer to another stack in a Parameter, so this is rarely necessary.

required_by: (optional) a list of other stacks or targets that require this stack. It's an inverse to `requires`.

tags: (optional) a dictionary of CloudFormation tags to apply to this stack. This will be combined with the global tags, but these tags will take precedence.

stack_name: (optional) If provided, this will be used as the name of the CloudFormation stack. Unlike `name`, the value doesn't need to be unique within the config, since you could have multiple stacks with the same name, but in different regions or accounts. (note: the namespace from the environment will be prepended to this)

stack_policy_path: (optional): If provided, specifies the path to a JSON formatted stack policy that will be applied when the CloudFormation stack is created and updated. You can use stack policies to prevent CloudFormation from making updates to protected resources (e.g. databases). See: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/protect-stack-resources.html

in_progress_behavior: (optional): If provided, specifies the behavior for when a stack is in `CREATE_IN_PROGRESS` or `UPDATE_IN_PROGRESS`. By default, CFNgin will raise an exception if the stack is in an `IN_PROGRESS` state. You can set this option to `wait` and CFNgin will wait for the previous update to complete before attempting to update the stack.

Stacks Example

Here's an example used to create a VPC:

```
stacks:
  - name: vpc-example
    class_path: blueprints.vpc.VPC
    locked: false
    enabled: true
    variables:
      InstanceType: t2.small
      SshKeyName: default
      ImageName: NAT
      AZCount: 2
      PublicSubnets:
        - 10.128.0.0/24
        - 10.128.1.0/24
        - 10.128.2.0/24
        - 10.128.3.0/24
      PrivateSubnets:
        - 10.128.8.0/22
        - 10.128.12.0/22
        - 10.128.16.0/22
        - 10.128.20.0/22
      CidrBlock: 10.128.0.0/16
```

Custom Log Formats

By default, CFNgin uses the following `log_formats`:

```
log_formats:
  info: "[%(asctime)s] %(message)s"
  color: "[%(asctime)s] \033[%(color)sm%(message)s\033[39m"
  debug: "[%(asctime)s] %(levelname)s %(threadName)s %(name)s:%(lineno)d(
↪%(funcName)s): %(message)s"
```

You may optionally provide custom *log_formats*. In this example, we add the environment name to each log line.

```
log_formats:
  info: "[%(asctime)s] ${environment} %(message)s"
  color: "[%(asctime)s] ${environment} \033[%(color)sm%(message)s\033[39m"
```

You may use any of the standard Python logging module format attributes when building your *log_formats*.

Variables

Variables are values that will be passed into a Blueprint before it is rendered. Variables can be any valid YAML data structure and can leverage Lookups to expand values at build time.

The following concepts make working with variables within large templates easier:

YAML anchors & references

If you have a common set of variables that you need to pass around in many places, it can be annoying to have to copy and paste them in multiple places. Instead, using a feature of YAML known as anchors & references, you can define common values in a single place and then refer to them with a simple syntax.

For example, say you pass a common domain name to each of your stacks, each of them taking it as a Variable. Rather than having to enter the domain into each stack (and hopefully not typo'ing any of them) you could do the following:

```
domain_name: &domain mydomain.com
```

Now you have an anchor called **domain** that you can use in place of any value in the config to provide the value **mydomain.com**. You use the anchor with a reference.

```
stacks:
  - name: vpc
    class_path: blueprints.vpc.VPC
    variables:
      DomainName: *domain
```

Even more powerful is the ability to anchor entire dictionaries, and then reference them in another dictionary, effectively providing it with default values.

```
common_variables: &common_variables
  DomainName: mydomain.com
  InstanceType: m3.medium
  AMI: ami-12345abc
```

Now, rather than having to provide each of those variables to every stack that could use them, you can just do this instead.

```
stacks:
  - name: vpc
    class_path: blueprints.vpc.VPC
    variables:
      << : *common_variables
      InstanceType: c4.xlarge # override the InstanceType in this stack
```

Using Outputs as Variables

Since CFNgin encourages the breaking up of your CloudFormation stacks into entirely separate stacks, sometimes you'll need to pass values from one stack to another. The way this is handled in CFNgin is by having one stack provide Outputs for all the values that another stack may need, and then using those as the inputs for another stack's *Variables*. CFNgin makes this easier for you by providing a syntax for *Variables* that will cause CFNgin to automatically look up the values of Outputs from another stack in its config. To do so, use the following format for the Variable on the target stack.

```
MyParameter: ${output OtherStack::OutputName}
```

Since referencing Outputs from stacks is the most common use case, `output` is the default lookup type. For more information see Lookups.

In this example config - when building things inside a VPC, you will need to pass the **VpcId** of the VPC that you want the resources to be located in. If the **vpc** stack provides an Output called **VpcId**, you can reference it easily.

```
domain_name: my_domain &domain

stacks:
  - name: vpc
    class_path: blueprints.vpc.VPC
    variables:
      DomainName: *domain
  - name: webservers
    class_path: blueprints.asg.AutoscalingGroup
    variables:
      DomainName: *domain
      VpcId: ${output vpc::VpcId} # gets the VpcId Output from the vpc stack
```

Note: Doing this creates an implicit dependency from the **webservers** stack to the **vpc** stack, which will cause CFNgin to submit the **vpc** stack, and then wait until it is complete until it submits the **webservers** stack.

Environments

A pretty common use case is to have separate environments that you want to look mostly the same, though with some slight modifications. For example, you might want a **production** and a **staging** environment. The production environment likely needs more instances, and often those instances will be of a larger instance type. Environments allow you to use your existing CFNgin config, but provide different values based on the environment file chosen on the command line. For more information, see the Environments documentation.

2.7.3 Environment Files

When using CFNgin, you can optionally provide an "environment" file. The CFNgin config file will be interpolated as a string.Template using the key/value pairs from the environment file. The format of the file is a single key/value per line, separated by a colon (**:**), like this:

```
vpcID: vpc-12345678
```

Provided the key/value vpcID above, you will now be able to use this in your configs for the specific environment you are deploying into. They act as keys that can be used in your config file, providing a sort of templating ability. This allows you to change the values of your config based on the environment you are in. For example, if you have a **webserver** stack, and you need to provide it a variable for the instance size it should use, you would have something like this in your config file.

```
stacks:
  - name: webservers
    class_path: blueprints.asg.AutoscalingGroup
    variables:
      InstanceType: m3.medium
```

But what if you needed more CPU in your production environment, but not in your staging? Without Environments, you'd need a separate config for each. With environments, you can simply define two different environment files with the appropriate **InstanceType** in each, and then use the key in the environment files in your config.

```
# in the file: stage.env
web_instance_type: m3.medium

# in the file: prod.env
web_instance_type: c4.xlarge

# in your config file:
stacks:
  - name: webservers
    class_path: blueprints.asg.AutoscalingGroup
    variables:
      InstanceType: ${web_instance_type}
```

2.7.4 Lookups

Note: Runway lookups and CFNgin lookups are not interchangeable. While they do share a similar base class and syntax, they exist in two different registries. Runway config files can't use CFNgin lookups just as the CFNgin config cannot use Runway lookups.

CFNgin provides the ability to dynamically replace values in the config via a concept called lookups. A lookup is meant to take a value and convert it by calling out to another service or system.

A lookup is denoted in the config with the `${<lookup type> <lookup input>}` syntax. If `<lookup type>` isn't provided, CFNgin will fall back to use the `output` lookup .

Lookups are only resolved within Variables. They can be nested in any part of a YAML data structure and within another lookup itself.

Note: If a lookup has a non-string return value, it can be the only lookup within a value.

ie. if `custom` returns a list, this would raise an exception:

```
Variable: ${custom something}, ${output otherStack::Output}
```

This is valid:

```
Variable: ${custom something}
```

For example, given the following:

```yaml
stacks:
  - name: sg
    class_path: some.stack.blueprint.Blueprint
    variables:
      Roles:
        - ${output otherStack::IAMRole}
      Values:
        Env:
          Custom: ${custom ${output otherStack::Output}}
          DBUrl: postgres://${output dbStack::User}@${output dbStack::HostName}
```

The Blueprint would have access to the following resolved variables dictionary:

```
# variables
{
  "Roles": ["other-stack-iam-role"],
  "Values": {
    "Env": {
      "Custom": "custom-output",
      "DBUrl": "postgres://user@hostname",
    },
  },
}
```

CFNgin includes the following lookup types:

- *output lookup*
- *ami lookup*
- *custom lookup*
- *default lookup*
- *dynamodb lookup*
- *envvar lookup*
- *file lookup*
- *hook_data lookup*
- *kms lookup*
- *rxref lookup*
- *ssmstore lookup*
- *xref lookup*

Output Lookup

The `output` lookup takes a value of the format: `<stack name>::<output name>` and retrieves the output from the given stack name within the current namespace.

CFNgin treats output lookups differently than other lookups by auto adding the referenced stack in the lookup as a requirement to the stack whose variable the output value is being passed to.

You can specify an output lookup with the following syntax:

```
ConfVariable: ${output someStack::SomeOutput}
```

default Lookup

The `default` lookup type will check if a value exists for the variable in the environment file, then fall back to a default defined in the CFNgin config if the environment file doesn't contain the variable. This allows defaults to be set at the config file level, while granting the user the ability to override that value per environment.

Format of value:

```
<env_var>::<default value>
```

Example

```
Groups: ${default app_security_groups::sg-12345,sg-67890}
```

If `app_security_groups` is defined in the environment file, its defined value will be returned. Otherwise, `sg-12345,sg-67890` will be the returned value.

Note: The `default` lookup only supports checking if a variable is defined in an environment file. It does not support other embedded lookups to see if they exist. Only checking variables in the environment file are supported. If you attempt to have the default lookup perform any other lookup that fails, CFNgin will throw an exception for that lookup and will stop your build before it gets a chance to fall back to the default in your config.

KMS Lookup

The `kms` lookup type decrypts its input value.

As an example, if you have a database and it has a parameter called `DBPassword` that you don't want to store in clear text in your config (maybe because you want to check it into your version control system to share with the team), you could instead encrypt the value using `kms`.

For example:

```
# We use the aws cli to get the encrypted value for the string
# "PASSWORD" using the master key called 'myKey' in us-east-1
$ aws --region us-east-1 kms encrypt --key-id alias/myKey \
    --plaintext "PASSWORD" --output text --query CiphertextBlob

CiD6bC8t2Y<...encrypted blob...>

# With CFNgin we would reference the encrypted value like:
```

(continues on next page)

(continued from previous page)

```
DBPassword: ${kms us-east-1@CiD6bC8t2Y<...encrypted blob...>}

# The above would resolve to
DBPassword: PASSWORD
```

This requires that the person using CFNgin has access to the master key used to encrypt the value.

It is also possible to store the encrypted blob in a file (useful if the value is large) using the `file://` prefix, ie:

```
DockerConfig: ${kms file://dockercfg}
```

Note: Lookups resolve the path specified with *file://* relative to the location of the config file, not where the CFNgin command is run.

XRef Lookup

The `xref` lookup type is very similar to the `output` lookup type, the difference being that `xref` resolves output values from stacks that aren't contained within the current CFNgin namespace, but are existing stacks containing outputs within the same region on the AWS account you are deploying into. `xref` allows you to lookup these outputs from the stacks already on your account by specifying the stacks fully qualified name in the CloudFormation console.

Where the `output` type will take a stack name and use the current context to expand the fully qualified stack name based on the namespace, `xref` skips this expansion because it assumes you've provided it with the fully qualified stack name already. This allows you to reference output values from any CloudFormation stack in the same region.

Also, unlike the `output` lookup type, `xref` doesn't impact stack requirements.

Example

```
ConfVariable: ${xref fully-qualified-stack::SomeOutput}
```

RXRef Lookup

The `rxref` lookup type is very similar to the `xref` lookup type, the difference being that `rxref` will lookup output values from stacks that are relative to the current namespace but external to the stack, but will not resolve them. `rxref` assumes the stack containing the output already exists.

Where the `xref` type assumes you provided a fully qualified stack name, `rxref`, like `output` expands and retrieves the output from the given stack name within the current namespace, even if not defined in the CFNgin config you provided it.

Because there is no requirement to keep all stacks defined within the same CFNgin YAML config, you might need the ability to read outputs from other stacks deployed by CFNgin into your same account under the same namespace. `rxref` gives you that ability. This is useful if you want to break up very large configs into smaller groupings.

Also, unlike the `output` lookup type, `rxref` doesn't impact stack requirements.

Example

```
# in example-us-east-1.env
namespace: MyNamespace

# in cfngin.yaml
ConfVariable: ${rxref my-stack::SomeOutput}

# the above would effectively resolve to
ConfVariable: ${xref MyNamespace-my-stack::SomeOutput}
```

Although possible, it is not recommended to use `rxref` for stacks defined within the same CFNgin YAML config.

File Lookup

The `file` lookup type allows the loading of arbitrary data from files on disk. The lookup additionally supports using a `codec` to manipulate or wrap the file contents prior to injecting it. The parameterized-b64 `codec` is particularly useful to allow the interpolation of CloudFormation parameters in a UserData attribute of an instance or launch configuration.

Basic examples:

```
# We've written a file to /some/path:
$ echo "hello there" > /some/path

# In CFNgin we would reference the contents of this file with the following
conf_key: ${file plain:file://some/path}

# The above would resolve to
conf_key: hello there

# Or, if we used wanted a base64 encoded copy of the file data
conf_key: ${file base64:file://some/path}

# The above would resolve to
conf_key: aGVsbG8gdGhlcmUK
```

Supported Codecs:

- **plain** - Load the contents of the file untouched. This is the only codec that should be used with raw Cloudformation templates (the other codecs are intended for blueprints).
- **base64** - Encode the plain text file at the given path with base64 prior to returning it
- **parameterized** - The same as plain, but additionally supports referencing CloudFormation parameters to create userdata that's supplemented with information from the template, as is commonly needed in EC2 UserData. For example, given a template parameter of BucketName, the file could contain the following text:

  ```
  #!/bin/sh
  aws s3 sync s3://{{BucketName}}/somepath /somepath
  ```

  and then you could use something like this in the YAML config file:

  ```
  UserData: ${file parameterized:/path/to/file}
  ```

  resulting in the UserData parameter being defined as:

```
{
    "Fn::Join" : [
        "",
        [
            "#!/bin/sh\naws s3 sync s3://",
            {
                "Ref" : "BucketName"
            },
            "/somepath /somepath"
        ]
    ]
}
```

- **parameterized-b64** - The same as parameterized, with the results additionally wrapped in `{"Fn::Base64": ...}`, which is what you actually need for EC2 UserData

  When using parameterized-b64 for UserData, you should use a local parameter defined as such.

```
from troposphere import AWSHelperFn

"UserData": {
    "type": AWSHelperFn,
    "description": "Instance user data",
    "default": Ref("AWS::NoValue")
}
```

  and then assign UserData in a LaunchConfiguration or Instance to `self.get_variables()["UserData"]`. Note that we use AWSHelperFn as the type because the parameterized-b64 codec returns either a Base64 or a GenericHelperFn troposphere object.

- **json** - Decode the file as JSON and return the resulting object
- **json-parameterized** - Same as `json`, but applying templating rules from `parameterized` to every object *value*. Note that object *keys* are not modified. Example (an external PolicyDocument):

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "some:Action"
            ],
            "Resource": "{{MyResource}}"
        }
    ]
}
```

- **yaml** - Decode the file as YAML and return the resulting object. All strings are returned as `unicode` even in Python 2.
- **yaml-parameterized** - Same as `json-parameterized`, but using YAML.

```
Version: 2012-10-17
Statement:
  - Effect: Allow
    Action:
      - "some:Action"
    Resource: "{{MyResource}}"
```

SSM Parameter Store Lookup

The `ssmstore` lookup type retrieves a value from the Simple Systems Manager Parameter Store.

As an example, if you have a database and it has a parameter called `DBUser` that you don't want to store in clear text in your config, you could instead store it as a SSM parameter named `MyDBUser`.

For example:

```
# We use the aws cli to store the database username
$ aws ssm put-parameter --name "MyDBUser" --type "String" \
    --value "root"

# In CFNgin we would reference the value like:
DBUser: ${ssmstore us-east-1@MyDBUser}

# Which would resolve to:
DBUser: root
```

Encrypted values ("SecureStrings") can also be used, which will be automatically decrypted (assuming the CFNgin user has access to the associated KMS key). Care should be taken when using this with encrypted values (i.e. a safe policy is to only use it with `no_echo` CFNString values)

The region can be omitted (e.g. `DBUser: ${ssmstore MyDBUser}`), in which case `us-east-1` will be assumed.

DynamoDb Lookup

The `dynamodb` lookup type retrieves a value from a DynamoDb table.

As an example, if you have a Dynamo Table named `TestTable` and it has an Item with a Primary Partition key called `TestKey` and a value named `BucketName` , you can look it up by using CFNgin. The lookup key in this case is TestVal

Example

```
# We can reference that dynamo value
BucketName: ${dynamodb us-east-1:TestTable@TestKey:TestVal.BucketName}

# Which would resolve to:
BucketName: test-bucket
```

You can lookup other data types by putting the data type in the lookup. Valid values are `S` (String), `N` (Number), `M` (Map), `L` (List).

Example

```
ServerCount: ${dynamodb us-east-1:TestTable@TestKey:TestVal.ServerCount[N]}
```

This would return an int value, rather than a string

You can lookup values inside of a map.

Example

```
ServerCount: ${dynamodb us-east-1:TestTable@TestKey:TestVal.ServerInfo[M].
                                                          ServerCount[N]}
```

Shell Environment Lookup

The `envvar` lookup type retrieves a value from a variable in the shell's environment.

Example:

```
# Set an environment variable in the current shell.
$ export DATABASE_USER=root

# In the CFNgin config we could reference the value:
DBUser: ${envvar DATABASE_USER}

# Which would resolve to:
DBUser: root
```

You can also get the variable name from a file, by using the `file://` prefix in the lookup, like so:

```
DBUser: ${envvar file://dbuser_file.txt}
```

EC2 AMI Lookup

The `ami` lookup is meant to search for the most recent AMI created that matches the given filters.

Valid arguments:

```
region OPTIONAL ONCE:
    e.g. us-east-1@

owners (comma delimited) REQUIRED ONCE:
    aws_account_id | amazon | self

name_regex (a regex) REQUIRED ONCE:
    e.g. my-ubuntu-server-[0-9]+

executable_users (comma delimited) OPTIONAL ONCE:
    aws_account_id | amazon | self
```

Any other arguments specified are sent as filters to the aws api For example, "architecture:x86_64" will add a filter.

Example:

```
# Grabs the most recently created AMI that is owned by either this account,
# amazon, or the account id 888888888888 that has a name that matches
# the regex "server[0-9]+" and has "i386" as its architecture.

# Note: The region is optional, and defaults to the current CFNgin region
ImageId: ${ami [<region>@]owners:self,888888888888,amazon name_regex:server[0-9]+␣
↪architecture:i386}
```

Hook Data Lookup

When using hooks, you can have the hook store results in the hook_data dictionary on the context by setting `data_key` in the hook config.

This lookup lets you look up values in that dictionary. A good example of this is when you use the aws_lambda hook to upload AWS Lambda code, then need to pass that code object as the **Code** variable in the aws_lambda blueprint dictionary.

Example

```
# If you set the ``data_key`` config on the aws_lambda hook to be "myfunction"
# and you name the function package "TheCode" you can get the troposphere
# awslambda.Code object with:

Code: ${hook_data myfunction::TheCode}
```

Custom Lookup

A custom lookup may be registered within the config. For more information see Configuring Lookups.

Writing A Custom Lookup

A custom lookup must be in an executable, importable python package or standalone file. The lookup must be importable using your current `sys.path`. This takes into account the sys_path defined in the config file as well as any `paths` of package_sources.

The lookup must be a class, preferable with a base class of `runway.cfngin.lookups.LookupHandler` with a `@classmethod` of handle that accepts four arguments - `value`, `context`, `provider`, `**kwargs`. There must be only one lookup per file. The file containing the lookup class must have a `TYPE_NAME` global variable with a value of the name that will be used to register the lookup.

The lookup must return a string if being used for a CloudFormation parameter.

If using boto3 in a lookup, use the `session_cache` instead of creating a new session to ensure the correct credentials are used.

```
"""Example lookup."""
from runway.lookups.base import LookupHandler
from runway.cfngin.context import Context
from runway.cfngin.providers.base import BaseProvider
from runway.cfngin.session_cache import get_session
from runway.cfngin.util import read_value_from_path

TYPE_NAME = 'mylookup'

class MylookupLookup(LookupHandler):
    """My lookup."""

    @classmethod
    def handle(cls,
               value: str,
               context: Context,
               provider: BaseProvider,
```

(continues on next page)

(continued from previous page)

```
            **kwargs: Any
            ) -> str:
        """Do something."""
        query, args = cls.parse(read_value_from_path(value))

        # example of using get_session for a boto3 session
        session = get_session(provider.region)
        s3_client = session.client('s3')

        return 'something'
```

2.7.5 Hooks

A hook is a python function or class method that is executed before or after the action is taken. To see how to define hooks in a config file see the Pre & Post Hooks documentation.

Built-in Hooks

aws_lambda.upload_lambda_functions

Description

Build Lambda payloads from user configuration and upload them to S3.

Constructs ZIP archives containing files matching specified patterns for each function, uploads the result to Amazon S3, then stores objects (of type `troposphere.awslambda.Code`) in the context's hook data, ready to be referenced in blueprints.

Configuration consists of some global options, and a dictionary of function specifications. In the specifications, each key indicating the name of the function (used for generating names for artifacts), and the value determines what files to include in the ZIP (see more details below).

If a `requirements.txt` or `Pipfile/Pipfile.lock` files are found at the root of the provided `path`, the hook will use the appropriate method to package dependencies with your source code automatically. If you want to explicitly use `pipenv` over `pip`, provide `use_pipenv: true` for the function.

Docker can be used to collect python dependencies instead of using system python to build appropriate binaries for Lambda. This can be done by including the `dockerize_pip` configuration option which can have a value of `true` or `non-linux`.

Payloads are uploaded to either a custom bucket or the CFNgin default bucket, with the key containing it's checksum, to allow repeated uploads to be skipped in subsequent runs.

Hook Path

`runway.cfngin.hooks.aws_lambda.upload_lambda_functions`

Args

bucket (Optional[str]) Custom bucket to upload functions to. Omitting it will cause the default CFNgin bucket to be used.

bucket_region (Optional[str]) The region in which the bucket should exist. If not given, the region will be either be that of the global `cfngin_bucket_region` setting, or else the region in use by the provider.

prefix (Optional[str]) S3 key prefix to prepend to the uploaded zip name.

follow_symlinks (Optional[bool]) Will determine if symlinks should be followed and included with the zip artifact. (*default:* `False`)

payload_acl (Optional[str]) The canned S3 object ACL to be applied to the uploaded payload. (*default: private*)

functions (Dict[str, Any]) Configurations of desired payloads to build. Keys correspond to function names, used to derive key names for the payload. Each value should itself be a dictionary, with the following data:

docker_file (Optional[str]) Path to a local DockerFile that will be built and used for `dockerize_pip`. Must provide exactly one of `docker_file`, `docker_image`, or `runtime`.

docker_image (Optional[str]) Custom Docker image to use with `dockerize_pip`. Must provide exactly one of `docker_file`, `docker_image`, or `runtime`.

dockerize_pip (Optional[Union[str, bool]]) Whether to use Docker when restoring dependencies with pip. Can be set to `true`/`false` or the special string `non-linux` which will only run on non Linux systems. To use this option Docker must be installed.

exclude (Optional[Union[str, List[str]]]) Pattern or list of patterns of files to exclude from the payload. If provided, any files that match will be ignored, regardless of whether they match an inclusion pattern.

Commonly ignored files are already excluded by default, such as `.git`, `.svn`, `__pycache__`, `*.pyc`, `.gitignore`, etc.

include (Optional[Union[str, List[str]]]) Pattern or list of patterns of files to include in the payload. If provided, only files that match these patterns will be included in the payload.

Omitting it is equivalent to accepting all files that are not otherwise excluded.

path (str) Base directory of the Lambda function payload content. If it not an absolute path, it will be considered relative to the directory containing the CFNgin configuration file in use.

Files in this directory will be added to the payload ZIP, according to the include and exclude patterns. If not patterns are provided, all files in this directory (respecting default exclusions) will be used.

Files are stored in the archive with path names relative to this directory. So, for example, all the files contained directly under this directory will be added to the root of the ZIP file.

python_path (Optional[str]) Absolute path to a python interpreter to use for `pip`/`pipenv` actions. If not provided, the current python interpreter will be used for `pip` and `pipenv` will be used from the current `$PATH`.

runtime (Optional[str]) Runtime of the AWS Lambda Function being uploaded. Used with `dockerize_pip` to automatically select the appropriate Docker image to use. Must provide exactly one of `docker_file`, `docker_image`, or `runtime`.

use_pipenv (Optional[bool]): Will determine if pipenv will be used to generate requirements.txt from an existing Pipfile. To use this option pipenv must be installed.

Example

Hook configuration

```yaml
pre_build:
  upload_functions:
    path: runway.cfngin.hooks.aws_lambda.upload_lambda_functions
    required: true
    enabled: true
    data_key: lambda
    args:
      bucket: custom-bucket
      follow_symlinks: true
      prefix: cloudformation-custom-resources/
      payload_acl: authenticated-read
      functions:
        MyFunction:
          path: ./lambda_functions
          dockerize_pip: non-linux
          use_pipenv: true
          runtime: python3.8
          include:
            - '*.py'
            - '*.txt'
          exclude:
            - '*.pyc'
            - test/
```

Blueprint Usage

```python
from troposphere.awslambda import Function
from runway.cfngin.blueprints.base import Blueprint

class LambdaBlueprint(Blueprint):
    def create_template(self):
        code = self.context.hook_data['lambda']['MyFunction']

        self.template.add_resource(
            Function(
                'MyFunction',
                Code=code,
                Handler='my_function.handler',
                Role='...',
                Runtime='python2.7'
            )
        )
```

build_staticsite.build

Description

Build static site. Used by the staticsite module type.

Hook Path

`runway.hooks.staticsite.build_staticsite.build`

Args

See staticsite module documentation for details.

cleanup_s3.purge_bucket

Description

Delete objects in bucket. Primarily used as a `pre_destroy` hook before deleting an S3 bucket.

Hook Path

`runway.hooks.cleanup_s3.purge_bucket`

Args

bucket_name (str) Name of the S3 bucket.

bucket_output_lookup (str) Value to pass to `runway.cfngin.lookups.handlers.output.OutputLookup` to retrieve an S3 bucket name.

bucket_rxref_lookup (str) Value to pass to `runway.cfngin.lookups.handlers.rxref.RxrefLookup` to retrieve an S3 bucket name.

bucket_xref_lookup (str) Value to pass to `runway.cfngin.lookups.handlers.xref.XrefLookup` to retrieve an S3 bucket name.

cleanup_ssm.delete_param

Description

Delete SSM parameter. Primarily used when an SSM parameter is created by a hook rather than CloudFormation.

Hook Path

`runway.hooks.cleanup_ssm.delete_param`

Args

parameter_name (str) Name of an SSM parameter.

command.run_command

Description

Run a custom command as a hook.

Hook Path

`runway.cfngin.hooks.command.run_command`

Args

command (Union[str, List[str]]) Command(s) to run.

capture (bool) If enabled, capture the command's stdout and stderr, and return them in the hook result. (*default:* `False`)

interactive (bool) If enabled, allow the command to interact with stdin. Otherwise, stdin will be set to the null device. (*default:* `False`)

ignore_status (bool) Don't fail the hook if the command returns a non-zero status. (*default:* `False`)

quiet (bool) Redirect the command's stdout and stderr to the null device, silencing all output. Should not be enabled if `capture` is also enabled. (*default:* `False`)

stdin (Optional[str]) String to send to the stdin of the command. Implicitly disables `interactive`.

env (Optional[Dict[str, str]]) Dictionary of environment variable overrides for the command context. Will be merged with the current environment.

****kwargs (Any)** Any other arguments will be forwarded to the `subprocess.Popen` function. Interesting ones include: `cwd` and `shell`.

Example

```
pre_build:
  command_copy_environment:
    path: runway.cfngin.hooks.command.run_command
    required: true
    enabled: true
    data_key: copy_env
    args:
      command: ['cp', 'environment.template', 'environment']
  command_git_rev_parse:
    path: runway.cfngin.hooks.command.run_command
```

(continues on next page)

(continued from previous page)

```
    required: true
    enabled: true
    data_key: get_git_commit
    args:
      command: ['git', 'rev-parse', 'HEAD']
      cwd: ./my-git-repo
      capture: true
  command_npm_install:
    path: runway.cfngin.hooks.command.run_command
    args:
      command: '`cd $PROJECT_DIR/project; npm install`'
      env:
        PROJECT_DIR: ./my-project
        shell: true
```

ecs.create_clusters

Description

Create ECS clusters.

Hook Path

`runway.cfngin.hooks.ecs.create_clusters`

Args

clusters (List[str]) Names of clusters to create.

iam.create_ecs_service_role

Description

Create ecsServiceRole, which has to be named exactly that currently.

http://docs.aws.amazon.com/AmazonECS/latest/developerguide/IAM_policies.html#service_IAM_role

Hook Path

`runway.cfngin.hooks.iam.create_ecs_service_role`

Args

role_name (str) Name of the role to create. (*default: ecsServiceRole*)

iam.ensure_server_cert_exists

Description

Ensure server cert exists.

Hook Path

`runway.cfngin.hooks.iam.ensure_server_cert_exists`

Args

cert_name (str) Name of the certificate that should exist.

prompt (bool) Whether to prompt to upload a certificate if one does not exist. (*default:* `True`)

keypair.ensure_keypair_exists

Description

Ensure a specific keypair exists within AWS. If the key doesn't exist, upload it.

Hook Path

`runway.cfngin.hooks.keypair.ensure_keypair_exists`

Args

keypair (str) Name of the key pair to create

ssm_parameter_name (Optional[str]) Path to an SSM store parameter to receive the generated private key, instead of importing it or storing it locally.

ssm_key_id (Optional[str]) ID of a KMS key to encrypt the SSM parameter with. If omitted, the default key will be used.

public_key_path (Optional[str]) Path to a public key file to be imported instead of generating a new key. Incompatible with the SSM options, as the private key will not be available for storing.

route53.create_domain

Description

Create a domain within route53.

Hook Path

```
runway.cfngin.hooks.route53.create_domain
```

Args

domain (str) Domain name for the Route 53 hosted zone to be created.

upload_staticsite.get_distribution_data

Description

Retrieve information about the CloudFront distribution. Used by the staticsite module type.

Hook Path

```
runway.hooks.staticsite.upload_staticsite.get_distribution_data
```

Args

See staticsite module documentation for details.

upload_staticsite.sync

Description

Sync static website to S3 bucket. Used by the staticsite module type.

Hook Path

```
runway.hooks.staticsite.upload_staticsite.sync
```

Args

See staticsite module documentation for details.

Writing A Custom Hook

A custom hook must be in an executable, importable python package or standalone file. The hook must be importable using your current `sys.path`. This takes into account the sys_path defined in the config file as well as any `paths` of package_sources.

The hook must accept a minimum of two arguments, `context` and `provider`. Aside from the required arguments, it can have any number of additional arguments or use `**kwargs` to accept anything passed to it. The values for these additional arguments come from the `args` key of the hook definition.

The hook must return `True` or a truthy object if it was successful. It must return `False` or a falsy object if it failed. This signifies to CFNgin whether or not to halt execution if the hook is `required`. If data is returned, it can be accessed by subsequent hooks, lookups, or Blueprints from the context object. It will be stored as `context.hook_data[data_key]` where `data_key` is the value set in the hook definition.

If using boto3 in a hook, use the `session_cache` instead of creating a new session to ensure the correct credentials are used.

```
"""session_cache example."""
from runway.cfngin.session_cache import get_session

def do_something(context, provider, **kwargs):
    """Do something."""
    session = get_session(provider.region)
    s3_client = session.client('s3')
```

Example Hook Function

local_path/hooks/my_hook.py

```
"""My hook."""


def do_something(context, provider, is_failure=True, **kwargs):
    """Do something."""
    if is_failure:
        return False
    return f"You are not a failure {kwargs.get('name', 'Kevin')}."
```

local_path/cfngin.yaml

```
namespace: example
sys_path: ./

hooks:
  my_hook_do_something:
    path: hooks.my_hook.do_something
    args:
      is_failure: False
```

Example Hook Class

local_path/hooks/my_hook.py

```
"""My hook."""

class MyClass:
    """My class."""

    SUCCESS_MESSAGE = 'You are not a failure {name}.'

    @classmethod
    def do_something(cls, context, provider, is_failure=True, **kwargs):
        """Do something."""
        if is_failure:
            return False
        return self.SUCCESS_MESSAGE.format(name=kwargs.get('name', 'Kevin'))
```

local_path/cfngin.yaml

```
namespace: example
sys_path: ./

hooks:
  my_hook_do_something:
    path: hooks.my_hook.MyClass.do_something
    args:
      is_failure: False
      name: Karen
```

2.7.6 Blueprints

Blueprints are python classes that dynamically build CloudFormation templates. Where you would specify a raw Cloudformation template in a stack using the `template_path` key, you instead specify a Blueprint python file using the `class_path` key.

Traditionally Blueprints are built using troposphere, but that is not absolutely necessary.

Making your own should be easy, and you can take a lot of examples from Runway blueprints. In the end, all that is required is that the Blueprint is a subclass of `runway.cfngin.blueprints.base` and it have the following methods:

```
# Initializes the Blueprint
def __init__(self, name, context, mappings=None):

# Updates self.template to create the actual template
def create_template(self):

# Returns a tuple: (version, rendered_template)
def render_template(self):
```

Variables

A Blueprint can define a `VARIABLES` property that defines the variables it accepts from the Config Variables.

`VARIABLES` should be a dictionary of `<variable name>: <variable definition>`. The variable definition should be a dictionary which supports the following optional keys:

type: The type for the variable value. This can either be a native python type or one of the *Variable Types*.

default: The default value that should be used for the variable if none is provided in the config.

description: A string that describes the purpose of the variable.

validator: An optional function that can do custom validation of the variable. A validator function should take a single argument, the value being validated, and should return the value if validation is successful. If there is an issue validating the value, an exception (`ValueError`, `TypeError`, etc) should be raised by the function.

no_echo: Only valid for variables whose type subclasses `CFNType`. Whether to mask the parameter value whenever anyone makes a call that describes the stack. If you set the value to true, the parameter value is masked with asterisks (*).

allowed_values: Only valid for variables whose type subclasses `CFNType`. The set of values that should be allowed for the CloudFormation Parameter.

allowed_pattern: Only valid for variables whose type subclasses `CFNType`. A regular expression that represents the patterns you want to allow for the CloudFormation Parameter.

max_length: Only valid for variables whose type subclasses `CFNType`. The maximum length of the value for the CloudFormation Parameter.

min_length: Only valid for variables whose type subclasses `CFNType`. The minimum length of the value for the CloudFormation Parameter.

max_value: Only valid for variables whose type subclasses `CFNType`. The max value for the CloudFormation Parameter.

min_value: Only valid for variables whose type subclasses `CFNType`. The min value for the CloudFormation Parameter.

constraint_description: Only valid for variables whose type subclasses `CFNType`. A string that explains the constraint when the constraint is violated for the CloudFormation Parameter.

Variable Types

Any native python type can be specified as the `type` for a variable. You can also use the following custom types:

TroposphereType

The `TroposphereType` can be used to generate resources for use in the Blueprint directly from user-specified configuration. Which case applies depends on what `type` was chosen, and how it would be normally used in the Blueprint (and CloudFormation in general).

Resource Types

When `type` is a Resource Type, the value specified by the user in the configuration file must be a dictionary, but with two possible structures.

When `many` is disabled, the top-level dictionary keys correspond to parameters of the `type` constructor. The key-value pairs will be used directly, and one object will be created and stored in the variable.

When `many` is enabled, the top-level dictionary *keys* are resource titles, and the corresponding *values* are themselves dictionaries, to be used as parameters for creating each of multiple `type` objects. A list of those objects will be stored in the variable.

Property Types

When `type` is a Property Type the value specified by the user in the configuration file must be a dictionary or a list of dictionaries.

When `many` is disabled, the top-level dictionary keys correspond to parameters of the `type` constructor. The key-value pairs will be used directly, and one object will be created and stored in the variable.

When `many` is enabled, a list of dictionaries is expected. For each element, one corresponding call will be made to the `type` constructor, and all the objects produced will be stored (also as a list) in the variable.

Optional variables

In either case, when `optional` is enabled, the variable may have no value assigned, or be explicitly assigned a null value. When that happens the variable's final value will be `None`.

Example

Below is an annotated example:

```
from runway.cfngin.blueprints.base import Blueprint
from runway.cfngin.blueprints.variables.types import TroposphereType
from troposphere import s3, sns

class Buckets(Blueprint):

    VARIABLES = {
        # Specify that Buckets will be a list of s3.Bucket types.
        # This means the config should a dictionary of dictionaries
        # which will be converted into troposphere buckets.
        "Buckets": {
            "type": TroposphereType(s3.Bucket, many=True),
            "description": "S3 Buckets to create.",
        },
        # Specify that only a single bucket can be passed.
        "SingleBucket": {
            "type": TroposphereType(s3.Bucket),
            "description": "A single S3 bucket",
        },
        # Specify that Subscriptions will be a list of sns.Subscription types.
        # Note: sns.Subscription is the property type, not the standalone
        # sns.SubscriptionResource.
```

(continues on next page)

(continued from previous page)

```
        "Subscriptions": {
            "type": TroposphereType(sns.Subscription, many=True),
            "description": "Multiple SNS subscription designations"
        },
        # Specify that only a single subscription can be passed, and that it
        # is made optional.
        "SingleOptionalSubscription": {
            "type": TroposphereType(sns.Subscription, optional=True),
            "description": "A single, optional SNS subscription designation"
        }
    }

    def create_template(self):
        t = self.template
        variables = self.get_variables()

        # The Troposphere s3 buckets have already been created when we
        # access variables["Buckets"], we just need to add them as
        # resources to the template.
        [t.add_resource(bucket) for bucket in variables["Buckets"]]

        # Add the single bucket to the template. You can use
        # `Ref(single_bucket)` to pass CloudFormation references to the
        # bucket just as you would with any other Troposphere type.
        # single_bucket = variables["SingleBucket"]
        t.add_resource(single_bucket)

        subscriptions = variables["Subscriptions"]
        optional_subscription = variables["SingleOptionalSubscription"]
        # Handle it in some special way...
        if optional_subscription is not None:
            subscriptions.append(optional_subscription)

        t.add_resource(sns.Topic(
            TopicName="one-test",
            Subscriptions=))

        t.add_resource(sns.Topic(
            TopicName="another-test",
            Subscriptions=subscriptions))
```

A sample config for the above:

```
stacks:
  - name: buckets
    class_path: path.to.above.Buckets
    variables:
      Buckets:
        # resource name (title) that will be added to CloudFormation.
        FirstBucket:
          # name of the s3 bucket
          BucketName: my-first-bucket
        SecondBucket:
          BucketName: my-second-bucket
      SingleBucket:
        # resource name (title) that will be added to CloudFormation.
        MySingleBucket:
```

(continues on next page)

(continued from previous page)

```
      BucketName: my-single-bucket
    Subscriptions:
      - Endpoint: one-lambda
        Protocol: lambda
      - Endpoint: another-lambda
        Protocol: lambda
    # The following could be omitted entirely
    SingleOptionalSubscription:
      Endpoint: a-third-lambda
      Protocol: lambda
```

CFNType

The `CFNType` can be used to signal that a variable should be submitted to CloudFormation as a Parameter instead of only available to the Blueprint when rendering. This is useful if you want to leverage AWS- Specific Parameter types (e.g. `List<AWS::EC2::Image::Id>`) or Systems Manager Parameter Store values (e.g. `AWS::SSM::Parameter::Value<String>`). See `runway.cfngin.blueprints.variables.types` for available subclasses of the `CFNType`.

Example

Below is an annotated example:

```
from runway.cfngin.blueprints.base import Blueprint
from runway.cfngin.blueprints.variables.types import (
    CFNString,
    EC2AvailabilityZoneNameList,
)


class SampleBlueprint(Blueprint):

    VARIABLES = {
        "String": {
            "type": str,
            "description": "Simple string variable",
        },
        "List": {
            "type": list,
            "description": "Simple list variable",
        },
        "CloudFormationString": {
            "type": CFNString,
            "description": "A variable which will create a CloudFormation Parameter␣
↪of type String",
        },
        "CloudFormationSpecificType": {
            "type": EC2AvailabilityZoneNameList,
            "description": "A variable which will create a CloudFormation Parameter␣
↪of type List<AWS::EC2::AvailabilityZone::Name>"
        },
    }
```

(continues on next page)

(continued from previous page)

```
    def create_template(self):
        t = self.template

        # `get_variables` returns a dictionary of <variable name>: <variablevalue>.
        # For the subclasses of `CFNType`, the values are
        # instances of `CFNParameter` which have a `ref` helper property
        # which will return a troposphere `Ref` to the parameter name.
        variables = self.get_variables()

        t.add_output(Output("StringOutput", variables["String"]))

        # variables["List"] is a native list
        for index, value in enumerate(variables["List"]):
            t.add_output(Output("ListOutput:{}".format(index), value))


        # `CFNParameter` values (which wrap variables with a `type`
        # that is a `CFNType` subclass) can be converted to troposphere
        # `Ref` objects with the `ref` property
        t.add_output(Output("CloudFormationStringOutput",
                            variables["CloudFormationString"].ref))
        t.add_output(Output("CloudFormationSpecificTypeOutput",
                            variables["CloudFormationSpecificType"].ref))
```

Utilizing Stack name within your Blueprint

Sometimes your Blueprint might want to utilize the already existing stack name within your Blueprint. CFNgin provides access to both the fully qualified stack name matching what's shown in the CloudFormation console, in addition to the stacks short name you have set in your YAML config.

Referencing Fully Qualified Stack name

The fully qualified name is a combination of the CFNgin namespace + the short name (what you set as `name` in your YAML config file). If your CFNgin namespace is `CFNginIsCool` and the stacks short name is `myAwesomeEC2Instance`, the fully qualified name would be:

```
CFNginIsCool-myAwesomeEC2Instance
```

To use this in your Blueprint, you can get the name from context using `self.context.get_fqn(self.name)`.

Referencing the Stack short name

The Stack short name is the name you specified for the stack within your YAML config. It does not include the namespace. If your CFNgin namespace is `CFNginIsCool` and the stacks short name is `myAwesomeEC2Instance`, the short name would be:

```
myAwesomeEC2Instance
```

To use this in your Blueprint, you can get the name from self.name: `self.name`

Example

Below is an annotated example creating a security group:

```
# we are importing Ref to allow for CFN References in the EC2 resource.  Tags
# will be used to set the Name tag
from troposphere import Ref, ec2, Tags
from runway.cfngin.blueprints.base import Blueprint
# CFNString is imported to allow for stand alone stack use
from runway.cfngin.blueprints.variables.types import CFNString

class SampleBlueprint(Blueprint):

  # VpcId set here to allow for Blueprint to be reused
  VARIABLES = {
  "VpcId": {
      "type": CFNString,
      "description": "The VPC to create the Security group in",
      }
  }


  def create_template(self):
      template = self.template
      # Assigning the variables to a variable
      variables = self.get_variables()
      # now adding a SecurityGroup resource named `SecurityGroup` to the CFN template
      template.add_resource(
        ec2.SecurityGroup(
          "SecurityGroup",
          # Referencing the VpcId set as the variable
          VpcId=variables['VpcId'].ref,
          # Setting the group description as the fully qualified name
          GroupDescription=self.context.get_fqn(self.name),
          # setting the Name tag to be the stack short name
          Tags=Tags(
            Name=self.name
            )
          )
        )
```

Testing Blueprints

When writing your own Blueprints its useful to write tests for them in order to make sure they behave the way you expect they would, especially if there is any complex logic inside.

To this end, a sub-class of the `unittest.TestCase` class has been provided: `runway.cfngin.blueprints.testutil.BlueprintTestCase`. You use it like the regular TestCase class, but it comes with an addition assertion: `assertRenderedBlueprint`. This assertion takes a Blueprint object and renders it, then compares it to an expected output, usually in `tests/fixtures/blueprints`.

Yaml (CFNgin) format tests

In order to wrap the `BlueprintTestCase` tests in a format similar to CFNgin's stack format, the `YamlDirTestGenerator` class is provided. When subclassed in a directory, it will search for yaml files in that directory with certain structure and execute a test case for it. As an example:

```
---
namespace: test
stacks:
  - name: test_stack
    class_path: cfngin_blueprints.s3.Buckets
    variables:
      var1: val1
```

When run from tests, this will create a template fixture file called test_stack.json containing the output from the `cfngin_blueprints.s3.Buckets` template.

2.7.7 Templates

CloudFormation templates can be provided via Blueprints or JSON/YAML. JSON/YAML templates are specified for stacks via the `template_path` config option (see Stacks).

Jinja2 Templating

Templates with a `.j2` extension will be parsed using Jinja2. The CFNgin `context` and `mappings` objects and stack `variables` objects are available for use in the template:

```
Description: TestTemplate
Resources:
  Bucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: {{ context.environment.foo }}-{{ variables.myparamname }}
```

2.8 Lookups

Runway Lookups allow the use of variables within the Runway config file. These variables can then be passed along to *deployments*, *modules* and *tests*.

The syntax for a lookup is `${<lookup-name> <query>::<arg-key>=<arg-value>}`

Component	Description
`${`	Signifies the opening of the lookup.
`<lookup-name>`	The name of the lookup you wish to use. (e.g. `env`)
`` `` ``	The separator between lookup name a query.
`<query>`	The value the lookup will be looking for. (e.g. `AWS_REGION`) | When using a lookup on a dictionary/mapping, like for the *var* lookup, you can get nested values by providing the full path to the value. (e.g. `ami.dev`}
`::`	The separator between a query and optional arguments.
`<arg-key>=<arg-value>`	An argument passed to a lookup. Multiple arguments can be passed to a lookup by separating them with a comma (`,`). Arguments are optional. Supported arguments depend on the lookup being used.

Lookups can be nested (e.g. `${var ami_id.${var AWS_REGION}}`).

Lookups can't resolve other lookups. For example, if i use `${var region}` in my Runway config file to resolve the `region` from my variables file, the value in the variables file can't be `${env AWS_REGION}`. Well, it can but it will resolve to the literal value provided, not an AWS region like you may expect.

2.8.1 Lookup Arguments

Arguments can be passed to Lookups to effect how they function.

To provide arguments to a Lookup, use a double-colon (`::`) after the query. Each argument is then defined as a **key** and **value** seperated with equals (=) and the arguments theselves are seperated with a comma (`,`). The arguments can have an optional space after the comma and before the next key to make them easier to read but this is not required. The value of all arguments are read as strings.

Example

```
${var my_query::default=true, transform=bool}
${env MY_QUERY::default=1,transform=bool}
```

Each Lookup may have their own, specific arguments that it uses to modify its functionality or the value it returns. There is also a common set of arguments that all Lookups accept.

Common Arguments

Argument	Description
`default`	If the Lookup is unable to find a value for the provided query, this value will be returned instead of raising a `ValueError`.
`transform`	Transform the data returned by a Lookup into a different datatype. Supports `str` and `bool`.

Example

```
deployments:
  - environments:
      some_variable: ${var some_value::default=my_value}
      comma_list: ${var my_list::default=undefined, transform=str}
```

2.8.2 Build-in Lookups

env

Retrieve a value from an environment variable.

The value is retrieved from a copy of the current environment variables that is saved to the context object. These environment variables are manipulated at runtime by Runway to fill in additional values such as `DEPLOY_ENVIRONMENT` and `AWS_REGION` to match the current execution.

Note: `DEPLOY_ENVIRONMENT` and `AWS_REGION` can only be resolved during the processing of a module. To ensure no error occurs when trying to resolve one of these in a *Deployment* definition, provide a default value.

If the lookup is unable to find an environment variable matching the provided query, the default value is returned or a `ValueError` is raised if a default value was not provided.

Example

```
deployment:
  - modules:
      - path: sampleapp.cfn
        environment:
          creator: ${env USER}
    env_vars:
      ENVIRONMENT: ${env DEPLOY_ENVIRONMENT::default=default}
```

var

Retrieve a variable from the variables file or definition.

If the lookup is unable to find an defined variable matching the provided query, the default value is returned or a `ValueError` is raised if a default value was not provided.

Nested values can be used by providing the full path to the value but, it will not select a list element.

The returned value can contain any YAML support data type (dictionaries/mappings/hashes, lists/arrays/sequences, strings, numbers, and booleon).

```
deployment:
  - modules:
      - path: sampleapp.cfn
        environment:
          ami_id: ${var ami_id.${env AWS_REGION}}
    env_vars:
      SOME_VARIABLE: ${var some_variable::default=default}
```

2.9 Defining Tests

2.9.1 Overview

Tests can be defined in the *runway config file* to test your *modules* in any way you desire before deploying. They are run by using the `runway test` *command*. *Tests* are run in the order they are defined.

Example:

```
tests:
  - name: example-test
    type: script
    args:
      commands:
        - echo "Success!"
```

Test Failures

The default behavior if one of the *tests* fails is to terminate execution. The subsequent commands will not be run and a non-zero exit code returned. This behavior can modified to continue testing and not result in a non-zero exit code on a per-test basis by adding `required: false` to the *test definition*.

Example:

```
tests:
  - name: hello-world
    type: script
    required: false
    args:
      commands:
        - echo "Hello World!"  && exit 1
```

2.9.2 Built-in Test Types

cfn-lint

Source: https://github.com/aws-cloudformation/cfn-python-lint

> *Validate CloudFormation yaml/json templates against the CloudFormation spec and additional checks. Includes checking valid values for resource properties and best practices.*

In order to use this *test*, there must be a `.cfnlintrc` file in the same directory as the *Runway config file*.

Example:

```
tests:
  - name: cfn-lint-example
    type: cfn-lint
```

script

Executes a list of provided commands. Each command is run in its own subprocess.

Commands are passed into the test using the `commands` argument.

Example:

```
tests:
  - name: hello-world
    type: script
    args:
      commands:
        - echo "Hello World!"
```

yamllint

Source: https://github.com/adrienverge/yamllint

> *A linter for YAML files. yamllint does not only check for syntax validity, but for weirdnesses like key repetition and cosmetic problems such as lines length, trailing spaces, indentation, etc.*

A `.yamllint` file can be placed at in the same directory as the *Runway config file* to customize the linter or, the Runway provided template will be used.

Example:

```
tests:
  - name: yamllint-example
    type: yamllint
```

2.10 Repo Structure

Projects deployed via Runway can be structured in a few ways; some examples follow:

2.10.1 Git Branches as Environments

This example shows two *modules* using environment git branches (these same files would be present in each environment branch, with changes to any environment promoted through branches):

```
.
├── myapp.cfn
│   ├── dev-us-west-2.env
│   ├── prod-us-west-2.env
│   ├── myapp.yaml
│   └── templates
│       └── foo.json
├── myapp.tf
│   ├── backend.tfvars
│   ├── dev-us-east-1.tfvars
│   ├── prod-us-east-1.tfvars
│   └── main.tf
└── runway.yml
```

2.10.2 Directories as Environments

The same two *modules* from the above *Git Branches as Environments* structure can instead be stored in a normal single-branch git repo. Each directory correlates with an environment (dev and prod in this example).

Environment changes are done by copying the environments' contents between each other. E.g., promotion from dev to prod could be as simple as `diff -u dev/ prod/` followed by `rsync -r --delete dev/ prod/`

Enabling that automated promotion is one of the reasons this example below has prod config files in the dev folder and vice versa. When promotions between environments are more hand managed, this is not technically required:

```
.
├── dev
│   ├── myapp.cfn
│   │   ├── dev-us-west-2.env
│   │   ├── prod-us-west-2.env
│   │   ├── myapp.yaml
│   │   └── templates
│   │       └── myapp_cf_template.json
│   ├── myapp.tf
│   │   ├── backend.tfvars
│   │   ├── dev-us-east-1.tfvars
│   │   ├── prod-us-east-1.tfvars
│   │   └── main.tf
│   └── runway.yml
└── prod
    ├── myapp.cfn
    │   ├── dev-us-west-2.env
    │   ├── prod-us-west-2.env
    │   ├── myapp.yaml
    │   └── templates
    │       └── myapp_cf_template.json
    ├── myapp.tf
    │   ├── backend.tfvars
    │   ├── dev-us-east-1.tfvars
    │   ├── prod-us-east-1.tfvars
    │   └── main.tf
    └── runway.yml
```

2.10.3 Directories as Environments with a Single Module

Another sample repo structure, showing environment folders containing a single CloudFormation *modules* at their root (using the `ignore_git_branch` *Runway config file* option and a single declared module of `./` to merge the Environment & Module folders).

See the *Directories as Environments* example above for more information on why this shows prod config files in the dev folder and vice versa:

```
.
├── dev
│   ├── dev-us-west-2.env
│   ├── prod-us-west-2.env
│   ├── myapp.yaml
│   ├── runway.yml
│   └── templates
│       └── myapp_cf_template.json
└── prod
    ├── dev-us-west-2.env
    ├── prod-us-west-2.env
    ├── myapp.yaml
    ├── runway.yml
    └── templates
        └── myapp_cf_template.json
```

2.11 Terminology

2.11.1 CFngin

Blueprint

A python class that is responsible for creating a CloudFormation template. Usually this is built using troposphere.

config

A YAML config file that defines the stack definitions for all of the stacks you want CFNgin to manage.

context

Context is responsible for translating the values passed in via the command line and specified in the *config* to stacks.

environment

A set of variables that can be used inside the config, allowing you to slightly adjust configs based on which environment you are launching.

graph

A mapping of **object name** to **set/list of dependencies**.

A graph is constructed for each execution of CFNgin from the contents of the *config* file.

Example

- **stack1** depends on nothing.
- **stack2** depends on **stack1**

hook

These are python functions/methods that are executed before or after the action is taken.

lookup

A method for expanding values in the *config* at build time. By default lookups are used to reference Output values from other stacks within the same *namespace*.

namespace

A way to uniquely identify a stack. Used to determine the naming of many things, such as the S3 bucket where compiled templates are stored, as well as the prefix for stack names.

output

A CloudFormation Template concept. Stacks can output values, allowing easy access to those values. Often used to export the unique ID's of resources that templates create. CFNgin makes it simple to pull outputs from one stack and then use them as a *variable* in another stack.

persistent graph

A *graph* that is persisted between CFNgin executions. It is stored in in the Stack S3 bucket.

provider

Provider that supports provisioning rendered blueprints. By default, an AWS provider is used.

stack

The resulting stack of resources that is created by CloudFormation when it executes a template. Each stack managed by CFNgin is defined by a *stack definition* in the *config*.

stack definition

Defines the *stack* you want to build, usually there are multiple of these in the *config*. It also defines the variables to be used when building the *stack*.

variable

Dynamic variables that are passed into stacks when they are being built. Variables are defined within the *config*.

2.12 Developer Guide

2.12.1 Getting Started

Before getting started, fork this repo and clone your fork.

Development Environment

This project uses `pipenv` to create Python virtual environment. This must be installed on your system before setting up your dev environment.

With pipenv installed, run `make sync_all` to setup your development environment. This will create all the requred virtual environments to work on runway, build docs locally, and run integration tests locally. The virtual environments all have Runway installed as editable meaning as you make changes to the code of your local clone, it will be reflected in all the virtual environments.

Branch Requirements

Branches must start with one of the following prefixes (e.g. `<prefix>/<your-branch-name>`). This is due to how labels are applied to PRs. If the branch does not meet the requirement, any PRs from it will be blocked from being merged.

bugfix | fix | hotfix The branch contains a fix for a big.

feature | feat The branch contains a new feature or enhancement to an existing feature.

docs | documentation The branch only contains updates to documentation.

maintain | maint | maintenance The branch does not contain changes to the project itself to is aimed at maintaining the repo, CI/CD, or testing infrastructure. (e.g. README, GitHub action, integration test infrastructure)

release Reserved for maintainers to prepare for the release of a new version.

PR Requirements

In order for a PR to be merged it must be passing all checks and be approved by at least one maintainer. Some of the checks can be run locally using `make lint` and `make test`.

To be considered for approval, the PR must meet the following requirements.

- Title must be a brief explanation of what was done in the PR (think commit message).
- A summary of was done.
- Explain why this change is needed.
- Detail the changes that were made (think CHANGELOG).
- Screenshot if applicable.
- Include tests for any new features or changes to existing features. (unit tests and integration tests depending on the nature of the change)
- Documentation was updated for any new feature or changes to existing features.

2.12.2 GitHub Actions

GitHub Actions are used to manage issues, pull requests, and releases.

Branch Name

Runs on PR open/reopen to check that the incoming branch is using one of the correct prefixes for labels to be applied.

Accepted Prefixes

- `bugfix/`
- `chore/`
- `docs/`
- `enhancement/`
- `feat/`
- `feature/`
- `fix/`
- `hotfix/`
- `maint/`
- `maintain/`
- `maintenance/`
- `release/`

Issue Management

Assigns first responders to a newly opened issue and applies initial labels of *status:review_required* and *priority:low* to denote that one of the first responders has not reviewed the issue yet and set initial triage level.

This will also try to identify if the issue is a feature request, bug report, or question based by looking for keywords and apply the appropriate label. The issue templates will result in the corresponding label being applied.

Release Management

When a commit is pushed to **release** (tag is pushed, PR is merged) a release draft is created (if one does not exist) and PRs since the last tag are added following the included template. Changes are categorized based on PR labels.

2.12.3 Building Pyinstaller Packages Locally

We use Pyinstaller to build executables that do not require Python to be installed on a system. These are built by Travis CI for distribution to ensure a consistent environment but they can also be build locally for testing.

Prerequisites

These need to be installed globally so they are not included in the Pipfile.

- `setuptools==45.2.0`
- `virtualenv==20.0.1`
- `pipenv==2018.11.26`

Process

1. Export `TRAVIS_OS_NAME` environment variable for your system (`linux`, `osx`, or `windows`).
2. Execute `make travisbuild_file` or `make travisbuild_folder` from the root of the repo.

The output of these commands can be found in `../artifacts`

2.12.4 Travis CI

If you would like to simulate a fully build/deploy of runway on your fork, you can do so by first signing up and Travis CI and linking it to your GitHub account. After doing so, there are a few environment variables that can be setup for your environment.

Travis CI Environment Variables

`AWS_ACCESS_KEY_ID`	Credentials required to deploy build artifacts to S3 at the end of the build stage. See below for permission requirements.
`AWS_BUCKET`	S3 bucket name where build artifacts will be pushed.
`AWS_BUCKET_PREFIX`	Prefix for all build artifacts published to S3.
`AWS_DEFAULT_REGION`	Region where S3 bucket is located.
`AWS_SECRET_ACCESS_KEY`	Credentials required to deploy build artifacts to S3 at the end of the build stage. See below for permission requirements.
`FORKED`	Used to enable the deploy steps in a forked repo.
`NPM_API_KEY`	API key from NPM.
`NPM_EMAIL`	Your email address tied to the API key.
`NPM_PACKAGE_NAME`	Name to use when publishing an npm package.
`NPM_PACKAGE_VERSION`	Override the version number used for npm.

Travis CI User Permissions Example

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:PutObjectVersionAcl",
                "s3:PutObjectTagging",
                "s3:PutObjectAcl",
                "s3:GetObject"
            ],
```

(continues on next page)

(continued from previous page)

```
            "Resource": "arn:aws:s3:::$BUCKET_NAME/$PREFIX/*"
        },
        {
            "Sid": "RequiredForCliSyncCommand",
            "Effect": "Allow",
            "Action": [
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::$BUCKET_NAME"
            ]
        }
    ]
}
```

2.13 Apache License

Version 2.0

Date January 2004

URL http://www.apache.org/licenses/

2.13.1 TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means *(i)* the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or *(ii)* ownership of fifty percent (50%) or more of the outstanding shares, or *(iii)* beneficial ownership of such entity.

"You" (or **"Your"**) shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the

Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License.

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License.

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution.

You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

- You must give any other recipients of the Work or Derivative Works a copy of this License; and
- You must cause any modified files to carry prominent notices stating that You changed the files; and
- You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
- If the Work includes a `"NOTICE"` text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such `NOTICE` file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a `NOTICE` text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the `NOTICE` file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the `NOTICE` text from the Work, provided that such additional attribution notices cannot be construed as modifying the License. You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions.

Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks.

This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the `NOTICE` file.

7. Disclaimer of Warranty.

Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an **"AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND**, either express or implied, including, without limitation, any warranties or conditions of **TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE**. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability.

In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability.

While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

runway.commands.deploy, 14
runway.commands.destroy, 15
runway.commands.dismantle, 16
runway.commands.envvars, 16
runway.commands.gen_sample, 16
runway.commands.init, 17
runway.commands.kbenv, 18
runway.commands.plan, 18
runway.commands.preflight, 19
runway.commands.run_aws, 19
runway.commands.run_python, 20
runway.commands.run_stacker, 20
runway.commands.takeoff, 20
runway.commands.taxi, 20
runway.commands.test, 20
runway.commands.tfenv, 21
runway.commands.whichenv, 21
runway.lookups.handlers.base, 92
runway.lookups.handlers.env, 92
runway.lookups.handlers.var, 93
runway.path.Path, 28
runway.runway_module_type.RunwayModuleType, 31

INDEX

C

Config (*class in runway.config*), 21

D

DeploymentDefinition (*class in runway.config*), 23

M

ModuleDefinition (*class in runway.config*), 25

R

runway.commands.deploy (*module*), 14
runway.commands.destroy (*module*), 15
runway.commands.dismantle (*module*), 16
runway.commands.envvars (*module*), 16
runway.commands.gen_sample (*module*), 16
runway.commands.init (*module*), 17
runway.commands.kbenv (*module*), 18
runway.commands.plan (*module*), 18
runway.commands.preflight (*module*), 19
runway.commands.run_aws (*module*), 19
runway.commands.run_python (*module*), 20
runway.commands.run_stacker (*module*), 20
runway.commands.takeoff (*module*), 20
runway.commands.taxi (*module*), 20
runway.commands.test (*module*), 20
runway.commands.tfenv (*module*), 21
runway.commands.whichenv (*module*), 21
runway.lookups.handlers.base (*module*), 92
runway.lookups.handlers.env (*module*), 92
runway.lookups.handlers.var (*module*), 93
runway.path.Path (*module*), 28
runway.runway_module_type.RunwayModuleType (*module*), 31

T

TestDefinition (*class in runway.config*), 31

V

VariablesDefinition (*class in runway.config*), 32