cfn-pyplates Documentation

Release 0.3.1

MetaMetrics, Inc.

August 17, 2014

Contents

1 Where to get it **3**

2 Documentation **5**

3 Intended Audience **7**

4 What is a pyplate? **9**

5 Features **11**

6 Contents **13**
- 6.1 Creating a CFN Template using Pyplates . . . 13
- 6.2 Advanced Usage . . . 40
- 6.3 API Reference . . . 44
- 6.4 Developer Guidelines . . . 51
- 6.5 cfn-pyplates . . . 53
- 6.6 Why use YAML for Options Mappings? . . . 54

7 Thanks **55**
- 7.1 MetaMetrics, Inc . . . 55
- 7.2 Contributors . . . 55

Python Module Index **57**

Amazon Web Services CloudFormation templates, generated with Python!

CHAPTER 1

Where to get it

- https://pypi.python.org/pypi/cfn-pyplates/
- easy_install cfn-pyplates
- pip install cfn-pyplates

CHAPTER 2

Documentation

- https://cfn-pyplates.readthedocs.org/

CHAPTER 3

Intended Audience

pyplates are intended to be used with the Amazon Web Services CloudFormation service. If you're already a CloudFormation (CFN) user, chances are good that you've already come up with fun and interesting ways of generating valid CFN templates. pyplates are a way to make those templates while leveraging all of the power that the python environment has to offer.

CHAPTER 4

What is a pyplate?

A pyplate is a class-based python representation of a JSON CloudFormation template and resources, with the goal of generating cloudformation templates based on input python templates (pyplates!) that reflect the cloudformation template hierarchy.

CHAPTER 5

Features

- Allows for easy customization of templates at runtime, allowing one pyplate to describe all of your CFN Stack roles (production, testing, dev, staging, etc).
- Lets you put comments right in the template!
- Supports all required elements of a CFN template, such as Parameters, Resources, Outputs, etc.)
- Supports all intrinsic CFN functions, such as base64, get_att, ref, etc.
- Converts intuitiviely-written python dictionaries into JSON templates, without having to worry about nesting or order-of-operations issues.

CHAPTER 6

Contents

6.1 Creating a CFN Template using Pyplates

6.1.1 As simple as it gets

Here's an example of the simplest pyplate you can make, which is one that defines a CloudFormationTemplate, and then adds one Resource to it. Let's say that this is "template.py" (a.k.a python template; a.k.a pyplate!)

CloudFormation won't let you make a stack with no Resources, so this template needs one. Notice how cft.resources is already there for you. In addition to CloudFormationTemplate, common things that you'll need are available right now without having to import them, including classes like Resource and Properties, as well as all of the intrinsic functions such as ref and base64.

You can see what the required properties of an AWS::EC2::Instance are here:

http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-properties-ec2-instance.html

template.py

```
# Start with a template...
cft = CloudFormationTemplate(description="A very small template.")

cft.resources.ec2_instance = Resource('AnInstance', 'AWS::EC2::Instance',
    Properties({
        # This is an ubuntu AMI, picked from http://cloud-images.ubuntu.com/
        # You may need to change this if you're not in the us-east-1 region
        # Or if Ubuntu deregisters the AMI
        'ImageID': 'ami-c30360aa',
        'InstanceType': 'm1.small',
    })
)
```

Now, on the command-line, we run `cfn_py_generate template.py out.json`. Here's what out.json looks like:

out.json

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "A very small template.",
```

```
  "Resources": {
    "AnInstance": {
      "Type": "AWS::EC2::Instance",
      "Properties": {
        "InstanceType": "m1.small",
        "ImageID": "ami-c30360aa"
      }
    }
  }
}
```

Upload that to CloudFormation to have it make you a stack of one unnamed instance.

6.1.2 But now let's do something useful

Okay, we've made an instance! It would be nice to actually hand it a user-data script, though. There are a few ways to go about that. Fortunately, the pyplate is written in python, and we can anything that python can do.

Note: Properties args can be either a 'Properties' object, or just a plain old python dict. For this example, we'll use a dict, but either way works.

---

template.py

```
cft = CloudFormationTemplate(description="A slightly more useful template.")

user_data_script = '''#!/bin/bash

echo "You can put your userdata script right here!"
'''

cft.resources.add(Resource('AnInstance', 'AWS::EC2::Instance',
    {
        'ImageId': 'ami-c30360aa',
        'InstanceType': 'm1.small',
        'UserData': base64(user_data_script),
    })
)
```

Alternatively, because this is python, you could put the userdata script in its own file, and read it in using normal file operations:

```
user_data_script = open('userdata.sh').read()
```

The output certainly makes a mess of the script file, but that's really a discussion between the JSON serializer and CloudFormation that we don't need to worry ourselves with. After, we're here because making proper JSON is not a task for a human. Writing python is much more appropriate.

```
cfn_py_generate template.py out.json
```

out.json

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
```

```
  "Description": "A slightly more useful template.",
  "Resources": {
    "AnInstance": {
      "Type": "AWS::EC2::Instance",
      "Properties": {
        "UserData": {
          "Fn::Base64": "#!/bin/bash\n\necho \"You can put your userdata script right here!\"\n"
        },
        "InstanceType": "m1.small",
        "ImageId": "ami-c30360aa"
      }
    }
  }
}
```

6.1.3 Adding Metadata and Other Attributes to Resources

Cloudformation provides extensive support for Metadata that may be used to associate structured data with a resource.

Note: AWS CloudFormation does not validate the JSON in the Metadata attribute.

Adding Metadata to an S3 bucket

s3.py

```
cft = CloudFormationTemplate(description="A slightly more useful template.")

cft.resources.add(
    Resource('MyS3Bucket', 'AWS::S3::Bucket', None, Metadata(
        {"Object1": "Location1", "Object2": "Location2"}
    ))
)
```

out.json

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "A slightly more useful template.",
  "Resources": {
    "MyS3Bucket": {
      "Type": "AWS::S3::Bucket",
      "Metadata": {
        "Object1": "Location1",
        "Object2": "Location2"
      }
    }
  }
}
```

Adding Metadata to an EC2 instance

ec2_instance.py

```
user_data_script = '''#!/bin/bash

echo "You can put your userdata script right here!"
'''

cft = CloudFormationTemplate(description="A slightly more useful template.")
properties = {
    'ImageId': 'ami-c30360aa',
    'InstanceType': 'm1.small',
    'UserData': base64(user_data_script),
}
attributes = [
    Metadata(
        {
            "AWS::CloudFormation::Init": {
                "config": {
                    "packages": {},
                    "sources": {},
                    "commands": {},
                    "files": {},
                    "services": {},
                    "users": {},
                    "groups": {}
                }
            }
        }
    ),
]

cft.resources.add(
    Resource('MyInstance', 'AWS::EC2::Instance', properties, attributes)
)
```

out.json

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "A slightly more useful template.",
  "Resources": {
    "MyInstance": {
      "Type": "AWS::EC2::Instance",
      "Properties": {
        "UserData": {
          "Fn::Base64": "#!/bin/bash\n\necho \"You can put your userdata script right here!\"\n"
        },
        "InstanceType": "m1.small",
        "ImageId": "ami-c30360aa"
      },
      "Metadata": {
        "AWS::CloudFormation::Init": {
          "config": {
            "files": {},
```

```
            "commands": {},
            "users": {},
            "sources": {},
            "groups": {},
            "services": {},
            "packages": {}
          }
        }
      }
    }
  }
}
```

Practical Metadata example for bootstrapping an instance

ec2_instance_attribs.py

```
user_data_script = '''#!/bin/bash

echo "You can put your userdata script right here!"
'''

cft = CloudFormationTemplate(description="A slightly more useful template.")
properties = {
    'ImageId': 'ami-c30360aa',
    'InstanceType': 'm1.small',
    'UserData': base64(user_data_script),
}
attributes = [
    Metadata(
        {
            "AWS::CloudFormation::Init": {
                "config": {
                    "packages": {
                        "rpm": {
                            "epel": "http://download.fedoraproject.org/pub/epel/5/i386/epel-release-5
                        },
                        "yum": {
                            "httpd": [],
                            "php": [],
                            "wordpress": []
                        },
                        "rubygems": {
                            "chef": ["0.10.2"]
                        }
                    },
                    "sources": {
                        "/etc/puppet": "https://github.com/user1/cfn-demo/tarball/master"
                    },
                    "commands": {
                        "test": {
                            "command": "echo \"$CFNTEST\" > test.txt",
                            "env": {"CFNTEST": "I come from config1."},
                            "cwd": "~",
                            "test": "test ! -e ~/test.txt",
                            "ignoreErrors": "false"
```

```
                    }
                },
                "files": {
                    "/tmp/setup.mysql": {
                        "content":
                            join('',
                                 "CREATE DATABASE ", ref("DBName"), ";\n",
                                 "CREATE USER '", ref("DBUsername"), "'@'localhost' IDENTIFIED BY
                                 ref("DBPassword"),
                                 "';\n",
                                 "GRANT ALL ON ", ref("DBName"), ".* TO '", ref("DBUsername"), "
                                 "FLUSH PRIVILEGES;\n"
                            ),
                        "mode": "000644",
                        "owner": "root",
                        "group": "root"
                    }
                },
                "services": {
                    "sysvinit": {
                        "nginx": {
                            "enabled": "true",
                            "ensureRunning": "true",
                            "files": ["/etc/nginx/nginx.conf"],
                            "sources": ["/var/www/html"]
                        },
                        "php-fastcgi": {
                            "enabled": "true",
                            "ensureRunning": "true",
                            "packages": {
                                "yum": ["php", "spawn-fcgi"]
                            }
                        },
                        "sendmail": {
                            "enabled": "false",
                            "ensureRunning": "false"
                        }
                    }
                },
                "users": {
                    "myUser": {
                        "groups": ["groupOne", "groupTwo"],
                        "uid": "50",
                        "homeDir": "/tmp"
                    }
                },
                "groups": {
                    "groupOne": {
                    },
                    "groupTwo": {
                        "gid": "45"
                    }
                }
            }
        }
    }
),
UpdatePolicy(
```

```
        {
            "AutoScalingRollingUpdate": {
                "MinInstancesInService": "1",
                "MaxBatchSize": "1",
                "PauseTime": "PT12M5S"
            }
        }
    ),
    DeletionPolicy("Retain"),
    DependsOn(ref("myDB"))
]
cft.resources.add(
    Resource('MyInstance', 'AWS::EC2::Instance', properties, attributes)
)
```

out.json

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "A slightly more useful template.",
  "Resources": {
    "MyInstance": {
      "Type": "AWS::EC2::Instance",
      "Properties": {
        "UserData": {
          "Fn::Base64": "#!/bin/bash\n\necho \"You can put your userdata script right here!\"\n"
        },
        "InstanceType": "m1.small",
        "ImageId": "ami-c30360aa"
      },
      "Metadata": {
        "AWS::CloudFormation::Init": {
          "config": {
            "files": {
              "/tmp/setup.mysql": {
                "content": {
                  "Fn::Join": [
                    "",
                    [
                      "CREATE DATABASE ",
                      {
                        "Ref": "DBName"
                      },
                      ";\n",
                      "CREATE USER '",
                      {
                        "Ref": "DBUsername"
                      },
                      "'@'localhost' IDENTIFIED BY '",
                      {
                        "Ref": "DBPassword"
                      },
                      "';\n",
                      "GRANT ALL ON ",
                      {
                        "Ref": "DBName"
```

```
            },
            ".* TO '",
            {
              "Ref": "DBUsername"
            },
            "'@'localhost';\n",
            "FLUSH PRIVILEGES;\n"
          ]
        ]
      },
      "owner": "root",
      "group": "root",
      "mode": "000644"
    }
  },
  "commands": {
    "test": {
      "test": "test ! -e ~/test.txt",
      "ignoreErrors": "false",
      "command": "echo \"$CFNTEST\" > test.txt",
      "cwd": "~",
      "env": {
        "CFNTEST": "I come from config1."
      }
    }
  },
  "users": {
    "myUser": {
      "uid": "50",
      "groups": [
        "groupOne",
        "groupTwo"
      ],
      "homeDir": "/tmp"
    }
  },
  "sources": {
    "/etc/puppet": "https://github.com/user1/cfn-demo/tarball/master"
  },
  "groups": {
    "groupTwo": {
      "gid": "45"
    },
    "groupOne": {}
  },
  "services": {
    "sysvinit": {
      "nginx": {
        "files": [
          "/etc/nginx/nginx.conf"
        ],
        "sources": [
          "/var/www/html"
        ],
        "ensureRunning": "true",
        "enabled": "true"
      },
      "sendmail": {
```

```
                "ensureRunning": "false",
                "enabled": "false"
              },
              "php-fastcgi": {
                "ensureRunning": "true",
                "packages": {
                  "yum": [
                    "php",
                    "spawn-fcgi"
                  ]
                },
                "enabled": "true"
              }
            }
          },
          "packages": {
            "rubygems": {
              "chef": [
                "0.10.2"
              ]
            },
            "yum": {
              "httpd": [],
              "php": [],
              "wordpress": []
            },
            "rpm": {
              "epel": "http://download.fedoraproject.org/pub/epel/5/i386/epel-release-5-4.noarch.rp
            }
          }
        }
      },
      "UpdatePolicy": {
        "AutoScalingRollingUpdate": {
          "PauseTime": "PT12M5S",
          "MaxBatchSize": "1",
          "MinInstancesInService": "1"
        }
      },
      "DeletionPolicy": "Retain",
      "DependsOn": {
        "Ref": "myDB"
      }
    }
  }
}
```

6.1.4 Referencing Other Template Objects

This is where things start to really come together. The instrinsic functions `ref` and `get_att` are critical tools for getting the most out of CloudFormation templates.

What if you're using CloudFormation to describe a stack of Resources, but the goal is to try a bunch of different AMIs? Entering the AMIs at stack creation is a good way to tackle that situation, and parameters are how you do it. Use `ref` to refer to the parameter from the Instance properties.

While we're here, we'd also like to prompt the user for what instance type they'd like to spawn, as well as a friendly name to put on the instance for EC2 API tools, like the AWS Console. These are also good uses for `ref`

We also want to put the instance's DNS name in the stack outputs so we see it when using CFN API tools, and maybe act on it with some automation later. In this case, `get_att` is right for the job.

Here are some examples:

template.py

```
cft = CloudFormationTemplate(description="A self-referential template.")

cft.parameters.add(Parameter('ImageId', 'String',
    {
        'Default': 'ami-c30360aa',
        'Description': 'Amazon Machine Image ID to use for the created instance',
        'AllowedPattern': 'ami-[a-f0-9]+',
        'ConstraintDescription': 'must start with "ami-" followed by lowercase hexidecimal characters
    })
)

cft.parameters.add(Parameter('InstanceName', 'String',
    {
        'Description': 'A name for the instance to be created',
    })
)

cft.parameters.add(Parameter('InstanceType', 'String',
    {
        'Default': 'm1.small',
    })
)

# Now the Resource definition is all refs, totally customizeable at
# stack creation
cft.resources.add(Resource('AnInstance', 'AWS::EC2::Instance',
    {
        'ImageId': ref('ImageId'),
        'InstanceType': ref('InstanceType'),
        'Tags': ec2_tags({
            'Name': ref('InstanceName'),
        })
    })
)

cft.outputs.add(Output('DnsName',
    get_att('AnInstance', 'PublicDnsName'),
    'The public DNS Name for AnInstance')
)
```

```
cfn_py_generate template.py out.json
```

out.json

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
```

```
  "Description": "A self-referential template.",
  "Parameters": {
    "ImageId": {
      "Default": "ami-c30360aa",
      "AllowedPattern": "ami-[a-f0-9]+",
      "Type": "String",
      "Description": "Amazon Machine Image ID to use for the created instance",
      "ConstraintDescription": "must start with \"ami-\" followed by lowercase hexidecimal characters
    },
    "InstanceName": {
      "Type": "String",
      "Description": "A name for the instance to be created"
    },
    "InstanceType": {
      "Default": "m1.small",
      "Type": "String"
    }
  },
  "Resources": {
    "AnInstance": {
      "Type": "AWS::EC2::Instance",
      "Properties": {
        "Tags": [
          {
            "Value": {
              "Ref": "InstanceName"
            },
            "Key": "Name"
          }
        ],
        "InstanceType": {
          "Ref": "InstanceType"
        },
        "ImageId": {
          "Ref": "ImageId"
        }
      }
    }
  },
  "Outputs": {
    "DnsName": {
      "Description": "The public DNS Name for AnInstance",
      "Value": {
        "Fn::GetAtt": [
          "AnInstance",
          "PublicDnsName"
        ]
      }
    }
  }
}
```

6.1.5 Using the Options Mapping

In the introduction, there was some talk of using one pyplate to easily describe similar stacks. For example, let's say you have a static website being hosted on any number instances running in an auto scaling group behind a load balancer. This website has a few known roles, including development, testing, and production.

Using the options mapping, you can specify different options for each of those roles, and then plug them into the pyplate when the stack template is generated, giving you a custom template for each stack role.

Options mappins are defined in *YAML*. Here are some examples of the options for each stack role:

mappings/development.yaml

```
# A development stack, so very few and very small instances
StackRole: development
AppServerAvailabilityZones:
    # us-east-1b and us-east-1c smell funny, skip those...
    - us-east-1a
    - us-east-1d
    - us-east-1e
# micros are cheap, woo!
AppServerInstanceType: m1.micro
# No more/no less than 1
AutoScalinggroupMinSize: 1
AutoScalingGroupMaxSize: 1
# Don't scale based on CPU alarms...
# Even though we can't because the max group size is 1
CloudWatchAlarmActionsEnabled: false
# SSH keypair to use, this is one that developers have
KeyPair: developers
```

mappings/testing.yaml

```
# A testing stack, so small instances but now it'll scale a little
StackRole: testing
AppServerAvailabilityZones:
    - us-east-1a
    - us-east-1d
    - us-east-1e
# We can test on smaller instances than we need in production
AppServerInstanceType: m1.small
# We're testing, so we'd love to make sure scaling works
AutoScalinggroupMinSize: 1
AutoScalingGroupMaxSize: 4
CloudWatchAlarmActionsEnabled: true
# Let's boot a buntu...
ImageId: ami-c30360aa
# SSH keypair to use, developers can get in here, just like before
KeyPair: developers
```

mappings/production.yaml

```
# A production stack, so big instances, lots of scaling!
StackRole: production
AppServerAvailabilityZones:
    - us-east-1a
    - us-east-1d
    - us-east-1e
AppServerInstanceType: m1.large
AutoScalinggroupMinSize: 4 # Need at least 4 running all the time...
```

```
AutoScalingGroupMaxSize: 100 # Really? 100? Also YAML lets you put comments here.
CloudWatchAlarmActionsEnabled: true
ImageId: ami-c30360aa
# Only admins have this SSH keypair
KeyPair: production
```

And here's the pyplate:

template.py

```
description = 'My static webapp {0} stack'.format(options['StackRole'])
cft = CloudFormationTemplate(description)

# Make a load balancer
cft.resources.add(Resource('LoadBalancer',
    'AWS::ElasticLoadBalancing::LoadBalancer',
    {
        'AvailabilityZones': options['AppServerAvailabilityZones'],
        'HealthCheck': {
            'HealthyThreshold': '2',
            'Interval': '30',
            'Target': 'HTTP:80/',
            'Timeout': '5',
            'UnhealthyThreshold': '2'
        },
        'Listeners': [
            {
                'LoadBalancerPort': '80',
                'Protocol': 'HTTP',
                'InstanceProtocol': 'HTTP',
                'InstancePort': '80',
                'PolicyNames': []
            },
        ],
    })
)

# Make a security group for the load balancer
cft.resources.add(Resource('ELBSecurityGroup',
    'AWS::EC2::SecurityGroup',
    {
        'GroupDescription': 'allow traffic from our app servers to the load balancer'
    })
)

# Put an ingress policy on the load balancer security group
cft.resources.add(Resource('ELBSecurityGroupIngressHTTP',
    'AWS::EC2::SecurityGroupIngress',
    {
        'GroupName': ref('ELBSecurityGroup'),
        'IpProtocol': 'tcp',
        'FromPort': '80',
        'ToPort': '80',
        'SourceSecurityGroupName': get_att('LoadBalancer', 'SourceSecurityGroup.GroupName'),
        'SourceSecurityGroupOwnerId': get_att('LoadBalancer', 'SourceSecurityGroup.OwnerAlias')
    })
)
```

```
# Let folks SSH up to the instance
cft.resources.add(Resource('SSHSecurityGroup',
    'AWS::EC2::SecurityGroup',
    {
        'GroupDescription': 'allows inbound SSH from all',
        'SecurityGroupIngress': {
            'IpProtocol': 'tcp',
            'CidrIp': '0.0.0.0/0',
            'FromPort': '22',
            'ToPort': '22'
        }
    })
)

# Make our auto scaling group
cft.resources.add(Resource('AppServerAutoScalingGroup',
    'AWS::AutoScaling::AutoScalingGroup',
    {
        'AvailabilityZones': options['AppServerAvailabilityZones'],
        'HealthCheckGracePeriod': 300,
        'HealthCheckType': 'ELB',
        'LaunchConfigurationName': ref('AppServerAutoScalingLaunchConfig'),
        'LoadBalancerNames': [ref('LoadBalancer')],
        'MaxSize': options['AutoScalingGroupMaxSize'],
        'MinSize': options['AutoScalinggroupMinSize'],
        'Tags': [{
            'Key': 'Name',
            'Value': options['StackRole'] + '-static-app-server',
            'PropagateAtLaunch': True,
        }]
    })
)
# Create the auto scaling group configuration for managing the server instances
cft.resources.add(Resource('AppServerAutoScalingLaunchConfig',
    'AWS::AutoScaling::LaunchConfiguration',
    {
        'ImageId': options['ImageId'],
        'InstanceType': options['AppServerInstanceType'],
        'KeyName': options['KeyPair'],
        'SecurityGroups': [
            ref('ELBSecurityGroup'),
            ref('SSHSecurityGroup'),
        ],
        # Another way to pass a user-data script,
        # looks better than the first example, but it's more tedious
        'UserData': base64(join('\n',
            '#!/bin/bash -v',
            '# do stuff...',
            '# ',
            '# Like maybe kick off cfnbootstrap, using all of the',
            '# AWS:CloudFormation::Init Metadata That we could have',
            '# put on our AutoScalingGroup',
            'exit 0',
        ))
    })
)

# Scale up policy for when the scale up alarm trips
```

```
cft.resources.add(Resource('AppServerScaleUpPolicy',
    'AWS::AutoScaling::ScalingPolicy',
    {
        'AdjustmentType': 'ChangeInCapacity',
        'AutoScalingGroupName': ref('AppServerAutoScalingGroup'),
        'Cooldown': '600',
        'ScalingAdjustment': '1'
    })
)

# Scale down policy for when the scale down alarm trips
cft.resources.add(Resource('AppServerScaleDownPolicy',
    'AWS::AutoScaling::ScalingPolicy',
    {
        'AdjustmentType': 'ChangeInCapacity',
        'AutoScalingGroupName': ref('AppServerAutoScalingGroup'),
        'Cooldown': '600',
        'ScalingAdjustment': '-1'
    })
)

# CloudWatch scale up alarm for triggering scale events
cft.resources.add(Resource('AppServerCPUAlarmHigh',
    'AWS::CloudWatch::Alarm',
    {
        'AlarmDescription': 'Scale up if average CPU usage of the AppServers stays above 75% for at
        'Dimensions': [{'Name': 'AutoScalingGroupName', 'Value': ref('AppServerAutoScalingGroup')}],
        'Namespace': 'AWS/EC2',
        'MetricName': 'CPUUtilization',
        'Unit': 'Percent',
        'Period': '60',
        'EvaluationPeriods': '5',
        'Statistic': 'Average',
        'ComparisonOperator': 'GreaterThanThreshold',
        'Threshold': '75',
        'ActionsEnabled': options['CloudWatchAlarmActionsEnabled'],
        'AlarmActions': [ref('AppServerScaleUpPolicy')]
    })
)

# CloudWatch scale down alarm for triggering scale events
cft.resources.add(Resource('AppServerCPUAlarmLow',
    'AWS::CloudWatch::Alarm',
    {
        'AlarmDescription': 'Scale down if average CPU usage of the AppServers stays below 25% for at
        'Dimensions': [{'Name': 'AutoScalingGroupName', 'Value': ref('AppServerAutoScalingGroup')}],
        'Namespace': 'AWS/EC2',
        'MetricName': 'CPUUtilization',
        'Unit': 'Percent',
        'Period': '60',
        'EvaluationPeriods': '5',
        'Statistic': 'Average',
        'ComparisonOperator': 'LessThanThreshold',
        'Threshold': '25',
        'ActionsEnabled': options['CloudWatchAlarmActionsEnabled'],
        'AlarmActions': [ref('AppServerScaleUpPolicy')]
    })
)
```

```
# Add the load balancer endpoint to our outputs
cft.outputs.add(
    Output('LoadBalancerEndpoint',
        get_att('LoadBalancer', 'DNSName')
    )
)
```

Notice that 'ImageId' is absent from the development options mapping. This will trigger a prompt for the user to fill in the blanks. This does two things:

- It gives you the ability to easily add options at runtime where appropriate
- It helps you spot typos in options names

In our case, the former reason is what we're after. Our developers love to boot all sorts of different AMIs and mess around, so it's easiest just to put in a new ID every time we generate a template. Here's what that prompt looks like:

```
Key "ImageId" not found in the supplied options mapping.
You can enter it now (or leave blank for None/null):
>
```

Generate the development template:

- `cfn_py_generate template.py development.json -o mappings/development.yaml`

The ami flavor du jour is `ami-deadbeef`, which I entered in the prompt. You can see how it was inserted into the development.json below.

Now, generate a new stack template based on each remaining role:

- `cfn_py_generate template.py testing.json -o mappings/testing.yaml`
- `cfn_py_generate template.py production.json -o mappings/production.yaml`

And here are the generated templates for CloudFormation:

development.json


```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "My static webapp development stack",
  "Resources": {
    "LoadBalancer": {
      "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
      "Properties": {
        "HealthCheck": {
          "HealthyThreshold": "2",
          "Interval": "30",
          "Target": "HTTP:80/",
          "Timeout": "5",
          "UnhealthyThreshold": "2"
        },
        "Listeners": [
          {
            "InstancePort": "80",
            "PolicyNames": [],
            "LoadBalancerPort": "80",
            "Protocol": "HTTP",
            "InstanceProtocol": "HTTP"
          }
```

```
      ],
      "AvailabilityZones": [
        "us-east-1a",
        "us-east-1d",
        "us-east-1e"
      ]
    }
  },
  "ELBSecurityGroup": {
    "Type": "AWS::EC2::SecurityGroup",
    "Properties": {
      "GroupDescription": "allow traffic from our app servers to the load balancer"
    }
  },
  "ELBSecurityGroupIngressHTTP": {
    "Type": "AWS::EC2::SecurityGroupIngress",
    "Properties": {
      "FromPort": "80",
      "GroupName": {
        "Ref": "ELBSecurityGroup"
      },
      "SourceSecurityGroupOwnerId": {
        "Fn::GetAtt": [
          "LoadBalancer",
          "SourceSecurityGroup.OwnerAlias"
        ]
      },
      "SourceSecurityGroupName": {
        "Fn::GetAtt": [
          "LoadBalancer",
          "SourceSecurityGroup.GroupName"
        ]
      },
      "ToPort": "80",
      "IpProtocol": "tcp"
    }
  },
  "SSHSecurityGroup": {
    "Type": "AWS::EC2::SecurityGroup",
    "Properties": {
      "SecurityGroupIngress": {
        "ToPort": "22",
        "IpProtocol": "tcp",
        "FromPort": "22",
        "CidrIp": "0.0.0.0/0"
      },
      "GroupDescription": "allows inbound SSH from all"
    }
  },
  "AppServerAutoScalingGroup": {
    "Type": "AWS::AutoScaling::AutoScalingGroup",
    "Properties": {
      "MinSize": 1,
      "Tags": [
        {
          "PropagateAtLaunch": true,
          "Value": "development-static-app-server",
          "Key": "Name"
```

```
        }
      ],
      "MaxSize": 1,
      "HealthCheckGracePeriod": 300,
      "LaunchConfigurationName": {
        "Ref": "AppServerAutoScalingLaunchConfig"
      },
      "AvailabilityZones": [
        "us-east-1a",
        "us-east-1d",
        "us-east-1e"
      ],
      "LoadBalancerNames": [
        {
          "Ref": "LoadBalancer"
        }
      ],
      "HealthCheckType": "ELB"
    }
  },
  "AppServerAutoScalingLaunchConfig": {
    "Type": "AWS::AutoScaling::LaunchConfiguration",
    "Properties": {
      "UserData": {
        "Fn::Base64": {
          "Fn::Join": [
            "\n",
            [
              "#!/bin/bash -v",
              "# do stuff...",
              "# ",
              "# Like maybe kick off cfnbootstrap, using all of the",
              "# AWS:CloudFormation::Init Metadata That we could have",
              "# put on our AutoScalingGroup",
              "exit 0"
            ]
          ]
        }
      },
      "KeyName": "developers",
      "SecurityGroups": [
        {
          "Ref": "ELBSecurityGroup"
        },
        {
          "Ref": "SSHSecurityGroup"
        }
      ],
      "InstanceType": "m1.micro",
      "ImageId": "ami-deadbeef"
    }
  },
  "AppServerScaleUpPolicy": {
    "Type": "AWS::AutoScaling::ScalingPolicy",
    "Properties": {
      "ScalingAdjustment": "1",
      "Cooldown": "600",
      "AutoScalingGroupName": {
```

```
        "Ref": "AppServerAutoScalingGroup"
      },
      "AdjustmentType": "ChangeInCapacity"
    }
  },
  "AppServerScaleDownPolicy": {
    "Type": "AWS::AutoScaling::ScalingPolicy",
    "Properties": {
      "ScalingAdjustment": "-1",
      "Cooldown": "600",
      "AutoScalingGroupName": {
        "Ref": "AppServerAutoScalingGroup"
      },
      "AdjustmentType": "ChangeInCapacity"
    }
  },
  "AppServerCPUAlarmHigh": {
    "Type": "AWS::CloudWatch::Alarm",
    "Properties": {
      "Dimensions": [
        {
          "Name": "AutoScalingGroupName",
          "Value": {
            "Ref": "AppServerAutoScalingGroup"
          }
        }
      ],
      "Namespace": "AWS/EC2",
      "ActionsEnabled": false,
      "MetricName": "CPUUtilization",
      "EvaluationPeriods": "5",
      "AlarmActions": [
        {
          "Ref": "AppServerScaleUpPolicy"
        }
      ],
      "AlarmDescription": "Scale up if average CPU usage of the AppServers stays above 75% for at l
      "Period": "60",
      "ComparisonOperator": "GreaterThanThreshold",
      "Statistic": "Average",
      "Threshold": "75",
      "Unit": "Percent"
    }
  },
  "AppServerCPUAlarmLow": {
    "Type": "AWS::CloudWatch::Alarm",
    "Properties": {
      "Dimensions": [
        {
          "Name": "AutoScalingGroupName",
          "Value": {
            "Ref": "AppServerAutoScalingGroup"
          }
        }
      ],
      "Namespace": "AWS/EC2",
      "ActionsEnabled": false,
      "MetricName": "CPUUtilization",
```

```
        "EvaluationPeriods": "5",
        "AlarmActions": [
          {
            "Ref": "AppServerScaleUpPolicy"
          }
        ],
        "AlarmDescription": "Scale down if average CPU usage of the AppServers stays below 25% for at
        "Period": "60",
        "ComparisonOperator": "LessThanThreshold",
        "Statistic": "Average",
        "Threshold": "25",
        "Unit": "Percent"
      }
    }
  },
  "Outputs": {
    "LoadBalancerEndpoint": {
      "Value": {
        "Fn::GetAtt": [
          "LoadBalancer",
          "DNSName"
        ]
      }
    }
  }
}
```

testing.json

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "My static webapp testing stack",
  "Resources": {
    "LoadBalancer": {
      "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
      "Properties": {
        "HealthCheck": {
          "HealthyThreshold": "2",
          "Interval": "30",
          "Target": "HTTP:80/",
          "Timeout": "5",
          "UnhealthyThreshold": "2"
        },
        "Listeners": [
          {
            "InstancePort": "80",
            "PolicyNames": [],
            "LoadBalancerPort": "80",
            "Protocol": "HTTP",
            "InstanceProtocol": "HTTP"
          }
        ],
        "AvailabilityZones": [
          "us-east-1a",
          "us-east-1d",
          "us-east-1e"
        ]
```

```
    }
  },
  "ELBSecurityGroup": {
    "Type": "AWS::EC2::SecurityGroup",
    "Properties": {
      "GroupDescription": "allow traffic from our app servers to the load balancer"
    }
  },
  "ELBSecurityGroupIngressHTTP": {
    "Type": "AWS::EC2::SecurityGroupIngress",
    "Properties": {
      "FromPort": "80",
      "GroupName": {
        "Ref": "ELBSecurityGroup"
      },
      "SourceSecurityGroupOwnerId": {
        "Fn::GetAtt": [
          "LoadBalancer",
          "SourceSecurityGroup.OwnerAlias"
        ]
      },
      "SourceSecurityGroupName": {
        "Fn::GetAtt": [
          "LoadBalancer",
          "SourceSecurityGroup.GroupName"
        ]
      },
      "ToPort": "80",
      "IpProtocol": "tcp"
    }
  },
  "SSHSecurityGroup": {
    "Type": "AWS::EC2::SecurityGroup",
    "Properties": {
      "SecurityGroupIngress": {
        "ToPort": "22",
        "IpProtocol": "tcp",
        "FromPort": "22",
        "CidrIp": "0.0.0.0/0"
      },
      "GroupDescription": "allows inbound SSH from all"
    }
  },
  "AppServerAutoScalingGroup": {
    "Type": "AWS::AutoScaling::AutoScalingGroup",
    "Properties": {
      "MinSize": 1,
      "Tags": [
        {
          "PropagateAtLaunch": true,
          "Value": "testing-static-app-server",
          "Key": "Name"
        }
      ],
      "MaxSize": 4,
      "HealthCheckGracePeriod": 300,
      "LaunchConfigurationName": {
        "Ref": "AppServerAutoScalingLaunchConfig"
```

```
      },
      "AvailabilityZones": [
        "us-east-1a",
        "us-east-1d",
        "us-east-1e"
      ],
      "LoadBalancerNames": [
        {
          "Ref": "LoadBalancer"
        }
      ],
      "HealthCheckType": "ELB"
    }
  },
  "AppServerAutoScalingLaunchConfig": {
    "Type": "AWS::AutoScaling::LaunchConfiguration",
    "Properties": {
      "UserData": {
        "Fn::Base64": {
          "Fn::Join": [
            "\n",
            [
              "#!/bin/bash -v",
              "# do stuff...",
              "# ",
              "# Like maybe kick off cfnbootstrap, using all of the",
              "# AWS:CloudFormation::Init Metadata That we could have",
              "# put on our AutoScalingGroup",
              "exit 0"
            ]
          ]
        }
      },
      "KeyName": "developers",
      "SecurityGroups": [
        {
          "Ref": "ELBSecurityGroup"
        },
        {
          "Ref": "SSHSecurityGroup"
        }
      ],
      "InstanceType": "m1.small",
      "ImageId": "ami-c30360aa"
    }
  },
  "AppServerScaleUpPolicy": {
    "Type": "AWS::AutoScaling::ScalingPolicy",
    "Properties": {
      "ScalingAdjustment": "1",
      "Cooldown": "600",
      "AutoScalingGroupName": {
        "Ref": "AppServerAutoScalingGroup"
      },
      "AdjustmentType": "ChangeInCapacity"
    }
  },
  "AppServerScaleDownPolicy": {
```

```
      "Type": "AWS::AutoScaling::ScalingPolicy",
      "Properties": {
        "ScalingAdjustment": "-1",
        "Cooldown": "600",
        "AutoScalingGroupName": {
          "Ref": "AppServerAutoScalingGroup"
        },
        "AdjustmentType": "ChangeInCapacity"
      }
    },
    "AppServerCPUAlarmHigh": {
      "Type": "AWS::CloudWatch::Alarm",
      "Properties": {
        "Dimensions": [
          {
            "Name": "AutoScalingGroupName",
            "Value": {
              "Ref": "AppServerAutoScalingGroup"
            }
          }
        ],
        "Namespace": "AWS/EC2",
        "ActionsEnabled": true,
        "MetricName": "CPUUtilization",
        "EvaluationPeriods": "5",
        "AlarmActions": [
          {
            "Ref": "AppServerScaleUpPolicy"
          }
        ],
        "AlarmDescription": "Scale up if average CPU usage of the AppServers stays above 75% for at l
        "Period": "60",
        "ComparisonOperator": "GreaterThanThreshold",
        "Statistic": "Average",
        "Threshold": "75",
        "Unit": "Percent"
      }
    },
    "AppServerCPUAlarmLow": {
      "Type": "AWS::CloudWatch::Alarm",
      "Properties": {
        "Dimensions": [
          {
            "Name": "AutoScalingGroupName",
            "Value": {
              "Ref": "AppServerAutoScalingGroup"
            }
          }
        ],
        "Namespace": "AWS/EC2",
        "ActionsEnabled": true,
        "MetricName": "CPUUtilization",
        "EvaluationPeriods": "5",
        "AlarmActions": [
          {
            "Ref": "AppServerScaleUpPolicy"
          }
        ],
```

```
        "AlarmDescription": "Scale down if average CPU usage of the AppServers stays below 25% for at
        "Period": "60",
        "ComparisonOperator": "LessThanThreshold",
        "Statistic": "Average",
        "Threshold": "25",
        "Unit": "Percent"
      }
    }
  },
  "Outputs": {
    "LoadBalancerEndpoint": {
      "Value": {
        "Fn::GetAtt": [
          "LoadBalancer",
          "DNSName"
        ]
      }
    }
  }
}
```

production.json

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "My static webapp production stack",
  "Resources": {
    "LoadBalancer": {
      "Type": "AWS::ElasticLoadBalancing::LoadBalancer",
      "Properties": {
        "HealthCheck": {
          "HealthyThreshold": "2",
          "Interval": "30",
          "Target": "HTTP:80/",
          "Timeout": "5",
          "UnhealthyThreshold": "2"
        },
        "Listeners": [
          {
            "InstancePort": "80",
            "PolicyNames": [],
            "LoadBalancerPort": "80",
            "Protocol": "HTTP",
            "InstanceProtocol": "HTTP"
          }
        ],
        "AvailabilityZones": [
          "us-east-1a",
          "us-east-1d",
          "us-east-1e"
        ]
      }
    },
    "ELBSecurityGroup": {
      "Type": "AWS::EC2::SecurityGroup",
      "Properties": {
        "GroupDescription": "allow traffic from our app servers to the load balancer"
```

```
    }
  },
  "ELBSecurityGroupIngressHTTP": {
    "Type": "AWS::EC2::SecurityGroupIngress",
    "Properties": {
      "FromPort": "80",
      "GroupName": {
        "Ref": "ELBSecurityGroup"
      },
      "SourceSecurityGroupOwnerId": {
        "Fn::GetAtt": [
          "LoadBalancer",
          "SourceSecurityGroup.OwnerAlias"
        ]
      },
      "SourceSecurityGroupName": {
        "Fn::GetAtt": [
          "LoadBalancer",
          "SourceSecurityGroup.GroupName"
        ]
      },
      "ToPort": "80",
      "IpProtocol": "tcp"
    }
  },
  "SSHSecurityGroup": {
    "Type": "AWS::EC2::SecurityGroup",
    "Properties": {
      "SecurityGroupIngress": {
        "ToPort": "22",
        "IpProtocol": "tcp",
        "FromPort": "22",
        "CidrIp": "0.0.0.0/0"
      },
      "GroupDescription": "allows inbound SSH from all"
    }
  },
  "AppServerAutoScalingGroup": {
    "Type": "AWS::AutoScaling::AutoScalingGroup",
    "Properties": {
      "MinSize": 4,
      "Tags": [
        {
          "PropagateAtLaunch": true,
          "Value": "production-static-app-server",
          "Key": "Name"
        }
      ],
      "MaxSize": 100,
      "HealthCheckGracePeriod": 300,
      "LaunchConfigurationName": {
        "Ref": "AppServerAutoScalingLaunchConfig"
      },
      "AvailabilityZones": [
        "us-east-1a",
        "us-east-1d",
        "us-east-1e"
      ],
```

```
      "LoadBalancerNames": [
        {
          "Ref": "LoadBalancer"
        }
      ],
      "HealthCheckType": "ELB"
    }
  },
  "AppServerAutoScalingLaunchConfig": {
    "Type": "AWS::AutoScaling::LaunchConfiguration",
    "Properties": {
      "UserData": {
        "Fn::Base64": {
          "Fn::Join": [
            "\n",
            [
              "#!/bin/bash -v",
              "# do stuff...",
              "# ",
              "# Like maybe kick off cfnbootstrap, using all of the",
              "# AWS:CloudFormation::Init Metadata That we could have",
              "# put on our AutoScalingGroup",
              "exit 0"
            ]
          ]
        }
      },
      "KeyName": "production",
      "SecurityGroups": [
        {
          "Ref": "ELBSecurityGroup"
        },
        {
          "Ref": "SSHSecurityGroup"
        }
      ],
      "InstanceType": "m1.large",
      "ImageId": "ami-c30360aa"
    }
  },
  "AppServerScaleUpPolicy": {
    "Type": "AWS::AutoScaling::ScalingPolicy",
    "Properties": {
      "ScalingAdjustment": "1",
      "Cooldown": "600",
      "AutoScalingGroupName": {
        "Ref": "AppServerAutoScalingGroup"
      },
      "AdjustmentType": "ChangeInCapacity"
    }
  },
  "AppServerScaleDownPolicy": {
    "Type": "AWS::AutoScaling::ScalingPolicy",
    "Properties": {
      "ScalingAdjustment": "-1",
      "Cooldown": "600",
      "AutoScalingGroupName": {
        "Ref": "AppServerAutoScalingGroup"
```

```
      },
      "AdjustmentType": "ChangeInCapacity"
    }
  },
  "AppServerCPUAlarmHigh": {
    "Type": "AWS::CloudWatch::Alarm",
    "Properties": {
      "Dimensions": [
        {
          "Name": "AutoScalingGroupName",
          "Value": {
            "Ref": "AppServerAutoScalingGroup"
          }
        }
      ],
      "Namespace": "AWS/EC2",
      "ActionsEnabled": true,
      "MetricName": "CPUUtilization",
      "EvaluationPeriods": "5",
      "AlarmActions": [
        {
          "Ref": "AppServerScaleUpPolicy"
        }
      ],
      "AlarmDescription": "Scale up if average CPU usage of the AppServers stays above 75% for at l
      "Period": "60",
      "ComparisonOperator": "GreaterThanThreshold",
      "Statistic": "Average",
      "Threshold": "75",
      "Unit": "Percent"
    }
  },
  "AppServerCPUAlarmLow": {
    "Type": "AWS::CloudWatch::Alarm",
    "Properties": {
      "Dimensions": [
        {
          "Name": "AutoScalingGroupName",
          "Value": {
            "Ref": "AppServerAutoScalingGroup"
          }
        }
      ],
      "Namespace": "AWS/EC2",
      "ActionsEnabled": true,
      "MetricName": "CPUUtilization",
      "EvaluationPeriods": "5",
      "AlarmActions": [
        {
          "Ref": "AppServerScaleUpPolicy"
        }
      ],
      "AlarmDescription": "Scale down if average CPU usage of the AppServers stays below 25% for at
      "Period": "60",
      "ComparisonOperator": "LessThanThreshold",
      "Statistic": "Average",
      "Threshold": "25",
      "Unit": "Percent"
```

```
        }
      }
    },
    "Outputs": {
      "LoadBalancerEndpoint": {
        "Value": {
          "Fn::GetAtt": [
            "LoadBalancer",
            "DNSName"
          ]
        }
      }
    }
}
```

6.1.6 Go forth, and pyplate

As you can see, things with pyplates can escalate quickly. Fortunately, with the help of the python interpreter, a little bit of YAML, and CloudFormation itself, crazy templates like the above don't have to be written purely in JSON, with no comments.

See any room for improvement? Fork this on GitHub!

https://github.com/seandst/cfn-pyplates

6.2 Advanced Usage

6.2.1 Refactoring your pyplates

At some point, probably when you start managing multiple projects with pyplates or possibly earlier if you have a penchant for clean code, you will want to be able to reuse your pyplates definitions. Fortunately, a pyplate is a standard python class, so refactoring is a relatively straightforward matter of creating useful superclasses (with a few minor gotchas that are easy to work around.

Let's say you have a pyplate that creates a stack with an EC2 instance and an S3 bucket. It might look like this:

project.py

```
cft = CloudFormationTemplate(description='My project template.')

cft.parameters.add(
    Parameter('EC2InstanceType', 'String',
        {
            'Default': 'm1.small',
            'Description': 'Instance type to use for created Server EC2 instance',
            'AllowedPattern': 'm3.[a-z]+',
            'ConstraintDescription': 'Must use one of the m3 instance types.',
        }
    )
)

cft.resources.add(
    Resource('Server', 'AWS::EC2::Instance',
```

```
        {
            'ImageId': options['AmiId'],
            'InstanceType': ref('EC2InstanceType')
        }
    )
)

cft.resources.add(
    Resource('StaticFiles', 'AWS::S3::Bucket', {'AccessControl: PublicRead'}, Metadata(
        {'Object1': 'Location1', 'Object2': 'Location2'}
    ))
)
```

To begin our refactoring, we can begin by extracting resource creation into a subclass of *CloudFormationTemplate*.

refactored.py

```
class ProjectTemplate(CloudFormationTemplate):

    def add_resources(self):
        self.add_server()
        self.add_bucket()

    def add_server(self):
        self.parameters.add(
            Parameter('EC2InstanceType', 'String',
                {
                    'Default': 'm1.small',
                    'Description': 'Instance type to use for created Server EC2 instance',
                    'AllowedPattern': 'm3.[a-z]+',
                    'ConstraintDescription': 'Must use one of the m3 instance types.',
                }
            )
        )
        self.resources.add(
            Resource('Server', 'AWS::EC2::Instance',
                {
                    'ImageId': 'ami-c30360aa',
                    'InstanceType': ref('EC2InstanceType')
                }
            )
        )

    def add_bucket(self):
        self.resource.add(
            Resource('StaticFiles', 'AWS::S3::Bucket', {'AccessControl: PublicRead'}, Metadata(
                {'Object1': 'Location1', 'Object2': 'Location2'}
            ))
        )

cft = ProjectTemplate(description='My project template.')
cft.add_resources()
```

We now instantiate ProjectTemplate instead of CloudFormationTemplate, and rather than messing with a bunch of attributes on our pyplate instance, we just call cft.add_resources() and we're done.

6.2.2 Creating reusable pyplates

This already looks nicer, but if we create a new project, we still have to copy and paste all this code into a new pyplate. We haven't saved any typing, and we haven't made refactoring any easier. For that, we need to pull common code into a module on our python path that all of our projects can access.

As we do this, though, we lose access to all the all of the pre-existing variables that pyplates give us. We can import most of them from `pyplates.core` (`CloudFormationTemplate`, `Resource`, `Parameter`, `Output`, and `MetaData`) or `pyplates.functions` (`ref`, `join`, `get_att`, and `base64`). `options` is handled differently. We need to pass that in to our template explicitly as the second argument, after the description. It will then be available within the class as `self.options`.

basetemplate.py

```
from pyplates import core, functions

class BaseTemplate(core.CloudFormationTemplate):

    def add_server(self):
        self.parameters.add(
            core.Parameter('EC2InstanceType', 'String',
                {
                    'Default': 'm1.small',
                    'Description': 'Instance type to use for created Server EC2 instance',
                    'AllowedPattern': 'm3.[a-z]+',
                    'ConstraintDescription': 'Must use one of the m3 instance types.',
                }
            )
        )
        self.resources.add(
            core.Resource('Server', 'AWS::EC2::Instance',
                {
                    'ImageId': self.options['AmiId'],
                    'InstanceType': functions.ref('EC2InstanceType')
                }
            )
        )

    def add_bucket(self):
        self.resource.add(
            core.Resource(
                'StaticFiles',
                'AWS::S3::Bucket',
                {'AccessControl: PublicRead'},
                Metadata({'Object1': 'Location1', 'Object2': 'Location2'})
            )
        )
```

We can now use this base template as a catalog of components that we might want to include in our projects. Projects can define their own subclasses and only use those components that are relevant to them.

Our usual project's pyplate now looks like this:

inheriting.py

```
import sys

# If our base template isn't on the PYTHONPATH already, we need to do this:
sys.path.append('../path/to/base/templates')

import basetemplate

class InheritingTemplate(basetemplate.BaseTemplate):
    def add_resources(self):
        self.add_server()
        self.add_bucket()

cft = InheritingTemplate("Our usual project", options)
cft.add_resources()
```

And if we want to create another project that requires an S3 bucket only, we can do so. We can even add a CORS configuration to this bucket while still leveraging the base template. Our pyplate is really just a collection of dictionaries (JSONableDicts, technically), so all we need to do is alter the right part of the dictionary using standard python.

altered.py

```
import sys

# If our base template isn't on the PYTHONPATH already, we need to do this:
sys.path.append('../path/to/base/templates')

import basetemplate

class AlteredTemplate(basetemplate.BaseTemplate):
    """This project only needs an S3 bucket, but no EC2 server."""

    def add_resources(self):
        self.add_bucket()

    def add_bucket(self):
        """This will add a bucket using the base template, and then add a custom CORS
        configuration to it."""

        super(AlteredTemplate, self).add_bucket()
        self.resources['StaticFiles']['Properties']['CorsConfiguration'] = {
            'CorsRules': [
                {
                    'AllowedHeaders': ['*'],
                    'AllowedMethods': ['GET'],
                    'AllowedOrigins': ['*'],
                }
            ]
        }

cft = AlteredTemplate("S3 Bucket Project", options)
cft.add_resources()
```

6.2.3 Going further

You may wish to go even further with your pyplate refactoring. This is python, so anything is possible. You can build a collection of reusable tools to create various resource types, and then build a an abstraction layer on top of that for creating related groups of resources that work together, such as an SQS Queue and an IAM User and a set of permissions to allow the user to access the queue. You could build mixins to organize those functional abstractions. You could build a framework for dynamically managing resource dependencies. The sky is the limit. pyplates is deliberately kept simple, so that building on top of it is easy.

If you do find new ways to get more mileage out of your pyplates usage, please let us know. We'd love to hear about it.

6.3 API Reference

6.3.1 cfn_pyplates

pyplates: CloudFormation templates, generated with python

See also:

- https://aws.amazon.com/cloudformation/
- https://cfn-pyplates.readthedocs.org/ (you might already be here)
- https://github.com/seandst/cfn-pyplates/

6.3.2 cfn_pyplates.cli

CLI Entry points for handy bins

Documentation for CLI methods defined in this file will be that method's usage information as seen on the command-line.

`cfn_pyplates.cli.`**`generate`**`()`

Generate CloudFormation JSON Template based on a Pyplate

Usage:

```
cfpy_generate <pyplate> [<outfile>] [-o/--options=<options_mapping>]
cfpy_generate (-h|--help)
cfpy_generate --version
```

Arguments:

pyplate Input pyplate file name

outfile File in which to place the compiled JSON template (if omitted or '-', outputs to stdout)

Options:

-o –options=<options_mapping> Input JSON or YAML file for options mapping exposed in the pyplate as "options_mapping"

-h –help This usage information

WARNING! Do not use pyplates that you haven't personally examined!

A pyplate is a crazy hybrid of JSON-looking python. exec is used to read the pyplate, so any code in there is going to run, even potentailly harmful things.

Be careful.

6.3.3 cfn_pyplates.core

Core functionality and all required components of a working CFN template.

These are all available without preamble in a pyplate's global namespace.

class `cfn_pyplates.core.`**`JSONableDict`**(*update_dict=None*, *name=None*)
A dictionary that knows how to turn itself into JSON

Parameters

- **update_dict** – A dictionary of values for prepopulating the JSONableDict at instantiation
- **name** – An optional name. If left out, the class's (or subclass's) name will be used.

The most common use-case of any JSON entry in a CFN Template is the `{"Name": {"Key1": "Value1", "Key2": Value2"} }` pattern. The significance of a JSONableDict's subclass name, or explicitly passing a 'name' argument is accomodating this pattern. All JSONableDicts have names.

To create the pyplate equivalent of the above JSON, contruct a JSONableDict accordingly:

```
JSONableDict({'Key1': 'Value1', 'Key2', 'Value2'}, 'Name'})
```

Based on `ordereddict.OrderedDict`, the order of keys is significant.

`add`(*child*)
Add a child node

Parameters **child** – An instance of JSONableDict

Raises AddRemoveError - `cfn_pyplates.exceptions.AddRemoveError`

`json`
Accessor to the canonical JSON representation of a JSONableDict

`remove`(*child*)
Remove a child node

Parameters **child** – An instance of JSONableDict

Raises AddRemoveError - `cfn_pyplates.exceptions.AddRemoveError`

`to_json`(**args*, ***kwargs*)
Thin wrapper around the `json.dumps()` method.

Allows for passing any arguments that json.dumps would accept to completely customize the JSON output if desired.

class `cfn_pyplates.core.`**`CloudFormationTemplate`**(*description=None*, *options=None*)
The root element of a CloudFormation template [1]

Takes an option description string in the constructor Comes pre-loaded with all the subelements CloudFormation can stand:

•Parameters

[1] http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-structure.html

- Mappings
- Resources
- Outputs

class cfn_pyplates.core.**Parameters**(*update_dict=None*, *name=None*)
The base Container for parameters used at stack creation [2]

Attached to a cfn_pyplates.core.CloudFormationTemplate

class cfn_pyplates.core.**Mappings**(*update_dict=None*, *name=None*)
The base Container for stack option mappings [3]

Note: Since most lookups can be done inside a pyplate using python, this is normally unused.

Attached to a cfn_pyplates.core.CloudFormationTemplate

class cfn_pyplates.core.**Resources**(*update_dict=None*, *name=None*)
The base Container for stack resources [4]

Attached to a cfn_pyplates.core.CloudFormationTemplate

class cfn_pyplates.core.**Outputs**(*update_dict=None*, *name=None*)
The base Container for stack outputs [5]

Attached to a cfn_pyplates.core.CloudFormationTemplate

class cfn_pyplates.core.**Properties**(*update_dict=None*, *name=None*)
A properties mapping [6], used by various CFN declarations

Can be found in:

- cfn_pyplates.core.Parameters
- cfn_pyplates.core.Outputs
- cfn_pyplates.core.Resource

Properties will be most commonly found in Resources

class cfn_pyplates.core.**Mapping**(*name*, *mappings=None*)
A CFN Mapping [2]

Used in the cfn_pyplates.core.Mappings container, a Mapping defines mappings used within the Cloudformation template and is not the same as a PyPlates options mapping.

More information for mapping options:

http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/concept-mappings.html

Parameters

- **name** – The unique name of the mapping to add
- **mappings** – The dictionary of mappings

class cfn_pyplates.core.**Resource**(*name*, *type*, *properties=None*, *attributes=[]*)
A generic CFN Resource [7]

[2] http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/parameters-section-structure.html
[3] http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/mappings-section-structure.html
[4] http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/resources-section-structure.html
[5] http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/outputs-section-structure.html
[6] http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/properties-section-structure.html
[7] http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-template-resource-type-ref.html

Used in the cfn_pyplates.core.Resources container.

All resources have a name, and most have a 'Type' and 'Properties' dict. Thus, this class takes those as arguments and makes a generic resource.

The 'name' parameter must follow CFN's guidelines for naming [6] The 'type' parameter must be one of these:

http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-template-resource-type-ref.html

The optional 'properties' parameter is a dictionary of properties as defined by the resource type, see documentation related to each resource type

Parameters

- **name** – The unique name of the resource to add
- **type** – The type of this resource
- **properties** – Optional properties mapping to apply to this resource, can be an instance of JSONableDict or just plain old dict
- **attributes** – Optional (on of 'DependsOn', 'DeletionPolicy', 'Metadata', 'UpdatePolicy' or a list of 2 or more)

class cfn_pyplates.core.**Parameter**(*name*, *type*, *properties=None*)
A CFN Parameter [4]

Used in the cfn_pyplates.core.Parameters container, a Parameter will be used when the template is processed by CloudFormation to prompt the user for any additional input.

More information for Parameter options:

http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/parameters-section-structure.html

Parameters

- **name** – The unique name of the parameter to add
- **type** – The type of this parameter
- **properties** – Optional properties mapping to apply to this parameter

class cfn_pyplates.core.**Output**(*name*, *value*, *description=None*)
A CFN Output [3]

Used in the cfn_pyplates.core.Outputs, an Output entry describes a value to be shown when describe this stack using CFN API tools.

More information for Output options can be found here:

http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/outputs-section-structure.html

Parameters

- **name** – The unique name of the output
- **value** – The value the output should return
- **description** – An optional description of this output

class cfn_pyplates.core.**DependsOn**(*policy=None*)
A CFN Output [3]

Used in the cfn_pyplates.core.Resource, The DependsOn attribute enables you to specify that the creation of a specific resource follows another

More information for DependsOn Attribute can be found here:

http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-attribute-dependson.html

Parameters **properties** – The unique name of the output

class cfn_pyplates.core.**DeletionPolicy**(*policy=None*)
A CFN Output [3]

Used in the cfn_pyplates.core.Resource, The DeletionPolicy attribute enables you to specify how AWS CloudFormation handles the resource deletion.

More information for DeletionPolicy Attribute can be found here:

http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-attribute-deletionpolicy.html

Parameters **properties** – The unique name of the output

class cfn_pyplates.core.**UpdatePolicy**(*properties=None*)
A CFN Output [3]

Used in the cfn_pyplates.core.Resource, The UpdatePolicy attribute enables you to specify how AWS CloudFormation handles rolling updates for a particular resource.

More information for UpdatePolicy Attribute can be found here:

http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-attribute-updatepolicy.html

Parameters **properties** – The unique name of the output

class cfn_pyplates.core.**Metadata**(*properties=None*)
A CFN Output [3]

Used in the cfn_pyplates.core.Resource,The Metadata attribute enables you to associate structured data with a resource. By adding a Metadata attribute to a resource, you can add data in JSON format to the resource declaration. In addition, you can use intrinsic functions (such as GetAtt and Ref), parameters, and pseudo parameters within the Metadata attribute to add those interpreted values.

More information for Metadata can be found here:

http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-attribute-metadata.html

Parameters **properties** – The unique name of the output

cfn_pyplates.core.**ec2_tags**(*tags*)
A container for Tags on EC2 Instances

Tags are declared really verbosely in CFN templates, but we have opportunites in the land of python to keep things a little more sane.

So we can turn the AWS EC2 Tags example from this:

```
"Tags": [
    { "Key" : "Role", "Value": "Test Instance" },
    { "Key" : "Application", "Value" : { "Ref" : "AWS::StackName"} }
]
```

Into something more like this:

```
EC2Tags({
    'Role': 'Test Instance',
    'Application': ref('StackName'),
})
```

Parameters **tags** – A dictionary of tags to apply to an EC2 instance

6.3.4 cfn_pyplates.exceptions

exception cfn_pyplates.exceptions.**AddRemoveError**(*message=None*, **args*)
Raised when attempting to attach weird things to a JSONableDict

Weird things, in this case, mean anything that isn't a JSONableDict

Parameters **message** – An optional message to package with the Error

exception cfn_pyplates.exceptions.**Error**(*message=None*, **args*)
Base exception class for cfn_pyplates

A namespaced Exception subclass with explicit 'message' support. Will be handled at template generation, with the message being delivered to the user.

Parameters

- **message** – An optional message to package with the Error
- **args** – Any number of optional arguments, to be used as subclasses see fit.

exception cfn_pyplates.exceptions.**IntrinsicFuncInputError**(*message=None*, **args*)
Raised when passing bad input values to an intrinsic function

Parameters **message** – An optional message to package with the Error

6.3.5 cfn_pyplates.functions

Python wrappers for CloudFormation intrinsic functions [8]

These are all available without preamble in a pyplate's global namespace.

These help make the pyplate look a little more like python than JSON, and can be ignored if you want to write the raw JSON directly. (But you don't want that, right? After all, that's why you're using pyplates.)

Notes:

- All "Return" values are a little midleading, in that what really gets returned is the JSON-ified CFN intrinsic function. When using these in pyplates, though, it's probably easiest to forget that and only be concerned with how CFN will handle this, hence the Return values.
- To avoid the issue of using the wrong types with functions that take sequences as input (join, select), argument unpacking is used. Therefore, pass the sequence elements one at a time, rather than the sequence itself, after passing the separator (for join) or index (for select).
- Using CloudFormation's Join function versus a pythonic 'string'.join allows you to use CFN's intrinsic functions inside a join statement. The pythonic string join method may literally interpret a call to an intrinsic function, causing the resulting JSON to be interpreted as a string and ignored by the CloudFormation template parser

cfn_pyplates.functions.**base64**(*value*)
The intrinsic function Fn::Base64 returns the Base64 representation of the input string.

This function is typically used to pass encoded data to Amazon EC2 instances by way of the UserData property.

Parameters **value** – The string value you want to convert to Base64

Returns The original string, in Base64 representation

cfn_pyplates.functions.**find_in_map**(*map_name*, *key*, *value*)
The intrinsic function Fn::FindInMap returns the value of a key from a mapping declared in the Mappings section.

[8] http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/intrinsic-function-reference.html

Parameters

- **map_name** – The logical name of the mapping declared in the Mappings section that contains the key-value pair.
- **key** – The name of the mapping key whose value you want.
- **value** – The value for the named mapping key.

Returns The map value.

cfn_pyplates.functions.**get_att**(*logical_name*, *attribute*)

The intrinsic function Fn:GetAtt returns the value of an attribute from a resource in the template.

param logical_name The logical name of the resource that contains the attribute you want.

param attribute The name of the resource-specific attribute whose value you want. See the resource's reference

page

[#cfn-resources] for details about the attributes available for that resource type.

returns The attribute value.

cfn_pyplates.functions.**get_azs**(*region=''*)

The intrinsic function Fn::GetAZs returns an array that lists all Availability Zones for the specified region.

Because customers have access to different Availability Zones, the intrinsic function Fn::GetAZs enables template authors to write templates that adapt to the calling user's access. This frees you from having to hard-code a full list of Availability Zones for a specified region.

Parameters **region** – The name of the region for which you want to get the Availability Zones. You can use the AWS::Region pseudo parameter to specify the region in which the stack is created. Specifying an empty string is equivalent to specifying AWS::Region.

Returns The list of Availability Zones for the region.

cfn_pyplates.functions.**join**(*sep*, **args*)

The intrinsic function Fn::Join appends a set of values into a single value, separated by the specified delimiter.

If a delimiter is the empty string, the set of values are concatenated with no delimiter.

Parameters

- **delimiter** – The value you want to occur between fragments. The delimiter will occur between fragments only. It will not terminate the final value.
- ***args** – Any number of values you want combined, passed as positional arguments

Returns The combined string.

cfn_pyplates.functions.**select**(*index*, **args*)

The intrinsic function Fn::Select returns a single object from a list of objects by index.

> **Warning:** Important Fn::Select does not check for null values or if the index is out of bounds of the array. Both conditions will result in a stack error, so you should be certain that the index you choose is valid, and that the list contains non-null values.

Parameters

- **index** – The index of the object to retrieve. This must be a value from zero to N-1, where N represents the number of elements in the array.

- ***args** – Any number of objects to select from, passed as positional arguments. None of the arguments can be `None`

Returns The selected object.

`cfn_pyplates.functions.`**`ref`**`(`*logical_name*`)`

The intrinsic function Ref returns the value of the specified parameter or resource.

When you are declaring a resource in a template and you need to specify another template resource by name, you can use the Ref to refer to that other resource. In general, Ref returns the name of the resource. For example, a reference to an AWS::AutoScaling::AutoScalingGroup returns the name of that Auto Scaling group resource.

For some resources, an identifier is returned that has another significant meaning in the context of the resource. An AWS::EC2::EIP resource, for instance, returns the IP address, and an AWS::EC2::Instance returns the instance ID.

Parameters **logical_name** – The logical name of the resource or parameter you want to dereference.

Returns

- **When you specify a parameter's logical name, it returns the value** of the parameter.
- **When you specify a resource's logical name, it returns a value** that you can typically use to refer to that resource.

Note: You can also use Ref to add values to Output messages.

6.4 Developer Guidelines

6.4.1 Pull Requests

Pull requests should:

- Conform to existing style in the project
- Add documentation for new functionality, where applicable
- Add unit test coverage for new functionality, where applicable
- *Lint the code*

6.4.2 Linting

This project adheres to just about everything in PEP8, with the exception of line length restrictions and some specifics around indentation.

Line Length

Since the primary mechanism for code submission and review is GitHub, we're allowed a little more freedom with regard to line length. The GitHub review panel is 100 characters wide, so the max line length for this project is similarly 100 characters.

Block Indentation

PEP8 does account for indentation of multi-line block statements, but only "officially" supports the following:

```
def a_very_long_function_name_that_takes_a_lot_of_arguments(will, need,
                                                            to, be,
                                                            indented,
                                                            like, this)
```

or...

```
def a_very_long_function_name_that_takes_a_lot_of_arguments(
        will, need, to, be, indented, like, this)
```

This would normally be disallowed, but can lead to better looking and more compact (but still quite readable) code:

```
def a_very_long_function_name_that_takes_a_lot_of_arguments(will, need
        to, be, indented, like, this)
    # Note the two indents on the second line of the block statement,
    # which clearly separates the block statement from the trailing
    # code.
```

Flake8

Flake8 is an excellent tool for basic code linting to check for adherence to style. It can be installed with `pip` or `easy_install`.

An example flake8.conf, reflecting this project's specific exceptions to PEP8:

```
[flake8]
ignore = E128
max-line-length = 100
```

6.4.3 Releases

Before posting a release

- Ensure that tests pass on your local machine, and after pushing ensure that the tests pass in the continuous integration system.
- Ensure that the sphinx docs can be generated without error (preferably without any warnings, too)
- Create a source distribution with *python setup.py sdist*. Install that source distribution into a newly created virtualenv (or other "clean" environment), verify that it installs properly
- Create and tag a release commit, with changes listed. The tagging format is 'v$VERSION', where $VERSION is the version string from `cfn_pyplates/__init__.py`.

Versions

This project adheres to Semantic Versioning. During the beta period, the *x* (major) version number is 0, and the *y* and *z* versions are intended to represent what the *x* and *y* numbers would mean, respectively.

Thus, with the *x* version set to 0, incrementing the *y* version signals possible backward-incompatible API changes that will require dependent code changes, while incrementing the *z* version signals backward-compatible changes, either new functionality in the API, bug fixes, or general improvements.

See also:

PEP 8 http://www.python.org/dev/peps/pep-0008/

Semantic Versioning http://semver.org/

6.5 cfn-pyplates

Amazon Web Services CloudFormation templates, generated with Python!

6.5.1 Where to get it

- https://pypi.python.org/pypi/cfn-pyplates/
- easy_install cfn-pyplates
- pip install cfn-pyplates

6.5.2 Documentation

- https://cfn-pyplates.readthedocs.org/

6.5.3 Intended Audience

pyplates are intended to be used with the Amazon Web Services CloudFormation service. If you're already a CloudFormation (CFN) user, chances are good that you've already come up with fun and interesting ways of generating valid CFN templates. pyplates are a way to make those templates while leveraging all of the power that the python environment has to offer.

6.5.4 What is a pyplate?

A pyplate is a class-based python representation of a JSON CloudFormation template and resources, with the goal of generating cloudformation templates based on input python templates (pyplates!) that reflect the cloudformation template hierarchy.

6.5.5 Features

- Allows for easy customization of templates at runtime, allowing one pyplate to describe all of your CFN Stack roles (production, testing, dev, staging, etc).
- Lets you put comments right in the template!
- Supports all required elements of a CFN template, such as Parameters, Resources, Outputs, etc.)
- Supports all intrinsic CFN functions, such as base64, get_att, ref, etc.
- Converts intuitiviely-written python dictionaries into JSON templates, without having to worry about nesting or order-of-operations issues.

6.6 Why use YAML for Options Mappings?

YAML's killer feature is that it allows comments alongside your config entries. When the options mapping is deciding critical things in your pyplate, it's nice to be able to explain why something is one way and not another. For example:

```
# The task scheduler breaks if more than one instance spawns right now
# We're working on it in ticket #1234, but for now just cap the group at 1
TaskSchedulerAutoScalingMaxSize: 1
```

In addition to comments, breaking options out into the mapping mean that you can potentially spend less time in the pyplate itself by implementing branching logic based on keys in the options mapping.

Earlier drafts of cfn-pyplates used JSON for the options mapping, but despite JSON's already lightweight markup compared to something like XML, nothing beats YAML for its simplicity and content/markup ratio.

See Also

- http://yaml.org
- http://en.wikipedia.org/wiki/YAML

CHAPTER 7

Thanks

7.1 MetaMetrics, Inc

The original development of this library was to streamline the deployment process at MetaMetrics. We've been using pyplates for a while now, and we always intended to get it out for others to use once we were able to make the time. Normally, that means that it would never get done.

MetaMetrics deserves special thanks for not just allowing this to be shared, but actively encouraging its publishing. If that sounds like a good place to work, that's because it *is* a good place to work.

http://www.metametricsinc.com/job-openings/

7.2 Contributors

Sean Myers <sean.dst@gmail.com> Main contributor, project "owner"

Jon Woodbury <jpwoodbu@mybox.org> Did a lot of practical work with pyplates; all of the examples in these docs are based on his early work with pyplates

GitHub Contributors https://github.com/seandst/cfn-pyplates/graphs/contributors

C

cfn_pyplates, 44
cfn_pyplates.cli, 44
cfn_pyplates.core, 45
cfn_pyplates.exceptions, 49
cfn_pyplates.functions, 49