gcdt Documentation

Release 0.1.436

gcdt

Dec 19, 2019

Contents

CHAPTER 1

Introduction

This userguide aims to make it easy for you to get started using the gcdt tools in your projects. gcdt helps you to code your infrastructure on AWS and put it under version control as infrastructure-as-code together with the implementation of your service. In this way you can fully automate your infrastructure and your service deployments.

gcdt provides tools for traditional compute services and also managed serverless computing services. gcdt was implemented internally at glomex.

glomex – The Global Media Exchange – is a provider of a global, open marketplace for premium video content as well as a technical service provider for the entire value chain of online video marketing.

gcdt and userguide are released under MIT License.

The gcdt userguide starts with this introduction, then provides an overview on gcdt like a guided tour on how gcdt is structured and what it offers to you. The following parts each covers one gcdt tool. The remaining parts go into more technical topics like developing gcdt or using it as library to build your own tools based on gcdt.

This user guide assumes that you know the AWS services you want to automate so we do not cover AWS services in great detail and instead point to relevant documentation. But even if you are starting out on AWS, gcdt will help you to quickly leave the AWS webconsole behind and to move towards infrastructure-as-code.

1.1 gcdt software version

This guide covers gcdt version '0.1.436'.

1.2 Related documents

This section aims to provide to you a list of related documents that will be useful to gain a detailed understanding about what the gcdt tool suite does. With this background you will be able to tap into the full potential of the gcdt tools.

1.3 Problem reporting instructions

Please use Github issues to report gcdt issues: gcdt issues. To check on the progress check out the gcdt project board: gcdt project board

glomex employess can get immediate user support via gcdt slack channel or just stop by the glomex SRE lab in Munich, Landsberger Straße 110 at P-02-041.

CHAPTER 2

Getting Started Guide

Welcome to the getting started guide of `gcdt`. In this guide we will cover what `gcdt` is and how to use it to create beautiful infrastructure as code (IaC):

- using AWS Cloudformation with `kumo`
- deploy and configure AWS Lambda with `ramuda`
- deploy your application with AWS Codedeploy scripts and `tenkai`
- deploy API Gateway and manage API keys with `yugen` All of these things you can do for different Environments (dev, stage, prod)

2.1 Infrastructure as code

Infrastructure as Code (IaC) is a type of IT infrastructure that teams can automatically provision through code, rather than using a manual GUI-centerd process. Through defining your infrastructure as code you can apply similar tools and workflows like when working with your application code like:

- version control
- test
- review
- share
- reuse (like creating test envs)
- audit

2.2 Installation

2.2.1 Install Python

First of all you need to have Python installed. Python should be 2.7 or higher

On MacOS try to use preinstalled Python

2.2.2 Install pip and virtualenv

virtualenv is a tool to create isolated Python environments. virtualenv creates a folder which contains all the necessary executables to use the packages that a Python project would need.

MacOS

```
$ sudo pip install virtualenv --upgrade
```

2.2.3 Install gcdt and gcdt plugins

Install gcdt

First of all you need to create virtualenv and activate it. We recommend create virtualenv in the same directory as a project, and add it to `.gitignore`. It's pretty easy.

```
$ cd <project-folder>
$ virtualenv venv
$ source venv/bin/activate
$ pip install pip --upgrade # we always should have latest pip version in our␣
↪virtualenv
```

gcdt needs some `gcdt-glugins` so you should install these together. `gcdt-glugins` are powerful tool to add features to `gcdt` without having to directly modify the `gcdt` core. The easiest way is to put the dependencies into a *requirements_gcdt.txt* file:

```
gcdt
gcdt-config-reader
gcdt-lookups
gcdt-bundler
gcdt-slack-integration
```

You can find more information about plugins in docs then

```
$ pip install -U -r requirements_gcdt.txt
```

To check that everything is good and `gcdt` installed just do:

```
$ gcdt version
gcdt version 0.1.418
gcdt plugins:
 * gcdt-config-reader version 0.0.11
 * gcdt-bundler version 0.0.27
 * glomex-config-reader version 0.0.18
```

(continues on next page)

(continued from previous page)

```
 * gcdt-slack-integration version 0.0.11
 * gcdt-lookups version 0.0.12
```

2.2.4 Setting the ENV environment variable

For almost all gcdt commands you need to set the `ENV` environment variable. `ENV` is required to recognize which config file to use (`gcdt_<env>.json`). Usually you have a different config file for each environment (dev, stage, prod). You need to set `ENV` before running any `gcdt` command.

The following command will use the `gcdt_dev.json` config file

```
$ export PYTHONIOENCODING=UTF-8
$ ENV=dev kumo list
...
```

Alternatively you can set the `ENV` variable for the duration of your terminal session. Set it like this:

```
$ export ENV=dev
$ kumo list
...
```

2.3 Kumo

Kumo is a tool which help you to manage and deploy your infrastructure. AWS Cloudformation uses helps you configure and manage your AWS infrastructure as code. In `kumo` we have our cloudformation templates generated by troposphere. With kumo you can easily create (and configure) infrastructure for different environments (and AWS accounts) like for example (dev, stage, prod).

2.3.1 Create your first stack with kumo

First of all you need to create two files:

- *cloudformation.py* - here you will describe your infrastructure using `troposphere`
- *gcdt_(dev|stage|prod)* - settings for your ENV in `json` format, needs to include all parameters for the cloudformation template + stack name. You should have separate config for each ENV.

Let's create a simple `gcdts_dev.json`*(please change all values according your AWS account)*:

```
{
  "kumo": {
    "stack": {
      "StackName": "gcdt-sample-stack"
    },
    "parameters": {
      "VPCId": "lookup:stack:<stack-name>:DefaultVPCId",
      "ScaleMinCapacity": "1",
      "ScaleMaxCapacity": "1",
      "InstanceType": "t2.micro",
      "DefaultInstancePolicyARN": "lookup:stack:<stack-name>:DefaultInstancePolicyARN
↪",
      "AMI": "lookup:secret:ops.prod.base_ami"
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

The values for `VPCId` and `DefaultInstancePolicyARN` are filled by by the `gcdt-lookups` which then will be used in the template. The `gcdt-lookups` plugin will search the outputs in the CloudFormation stack (as mentioned in the config).

Instead of `<stack-name>` you should provide your stack name or use hardcoded value(not recommended). It's time to create our first Infrastructure as Code. Let's do this. Here is a simple cloudformation.py script. Use it as a template for creating your infrastructure.

2.3.2 Deploy stack to AWS

Before running a deployment we need to set some necessary ENV variables. **Remember**: You need this ENV variables exported each time before running any `gcdt` command.

```
$ export ENV=dev
$ export AWS_DEFAULT_PROFILE=glomex # Default profile. Generated by aws-mfa
$ export AWS_DEFAULT_REGION=eu-west-1
$ export PYTHONIOENCODING=UTF-8
```

Run your first infrastructure deployment. It's really easy:

```
$ kumo deploy
```

Kumo deploy output **More information about `kumo` can be found in docs**

2.4 Ramuda

Ramuda will help you to deploy, manage and control AWS Lambda. Runtimes supported by `ramuda` are: **nodejs4.3, nodejs6.10, python2.7, python3.6**

2.4.1 Deploy simple AWS Lambda

Create a `gcdt_(dev|stage|prod)` file or update if you created it with `kumo` *(please change all values according your AWS account)*:

```
"ramuda": {
  "bundling": {
    "folders": [
        {
            "source": "./node_modules",
            "target": "./node_modules"
        }
    ],
    "zip": "bundle.zip"
  },
  "lambda": {
    "name" = "jenkins-gcdt-lifecycle-for-ramuda",
    "description" = "lambda test for ramuda",
```

(continues on next page)

(continued from previous page)

```
    "role" = "lookup:stack:<stack-name>:LambdaArnForDeploy",
    "handlerFunction" = "handler.handle",
    "handlerFile" = "handler.py",
    "timeout" = "300",
    "memorySize" = "256",
    "vpc": {
            "subnetIds": [
                "lookup:stack:<stack-name>:LambdaSubnetIda",
                "lookup:stack:<stack-name>:LambdaSubnetIdb",
                "lookup:stack:<stack-name>:LambdaSubnetIdc"
            ],
        }
  }
}
```

then do:

```
$ ramuda deploy
```

More information about `ramuda` can be found in docs

2.5 Tenkai

tenkai will help you to deploy your application using AWS Codedeploy. tenkai will create an application bundle file and upload it to s3 with all files that you have in your `codedeploy` folder. Create or update the `gcdt_(dev|stage|prod)` file *(please change all values according your AWS account)*:

```
"tenkai": {
  "codedeploy": {
    "applicationName": "lookup:stack:gcdt-sample-stack:applicationName",
    "deploymentGroupName": "lookup:stack:gcdt-sample-stack:DeploymentGroup",
    "deploymentConfigName": "lookup:stack:gcdt-sample-stack:DeploymentConfig",
    "artifactsBucket": "lookup:stack:<stack-name>:s3DeploymentBucket"
  }
}
```

then do:

```
$ tenkai deploy
```

More information about `tenkai` can be found in docs

2.6 Yugen

yugen is a tool that will help you to deploy and manage your API with AWS API Gateway. All you need is to put your `swagger.yml` into the same folder as a `gcdt_(dev|stage|prod)` file. Also, add some new configs into it *(please change all values according your AWS account)*:

```
"yugen": {
    "api": {
        "apiKey": "xxxxxxxxxxxxxx",
        "description": "Gcdt sample API based on dp api-mock",
```

(continues on next page)

(continued from previous page)

```
        "name": "jenkins-gcdt-sample-api-dev",
        "targetStage": "mock"
    }
}
```

More information about `yugen` can be found in docs

CHAPTER 3

Overview

Outline how gcdt works at a high level

- Identify the key functions
- Inputs and Outputs - identify inputs what you, the reader, need to enter
- and outputs, what you expect in response for example reports
- provide further details
- assumptions on user experience, for examples, experience or proficiency in related areas or previous training

CHAPTER 4

Installing gcdt

This chapter covers the gcdt installation. gcdt's behaviour can be customized using plugins. The gcdt plugin mechanism relies on standard Python package mechanisms. In order to get a good experience and get the most out of gcdt you need to know a few things about Python packaging.

This chapter aims to provide you with all the information you need to know on this topic.

4.1 Related documents

4.2 What you need to know about python package management

There is now a lot of packaging infrastructure in the Python community, a lot of technology, and a lot of experience. We will try cover some basic things and give you best practice what to use for Python package management.

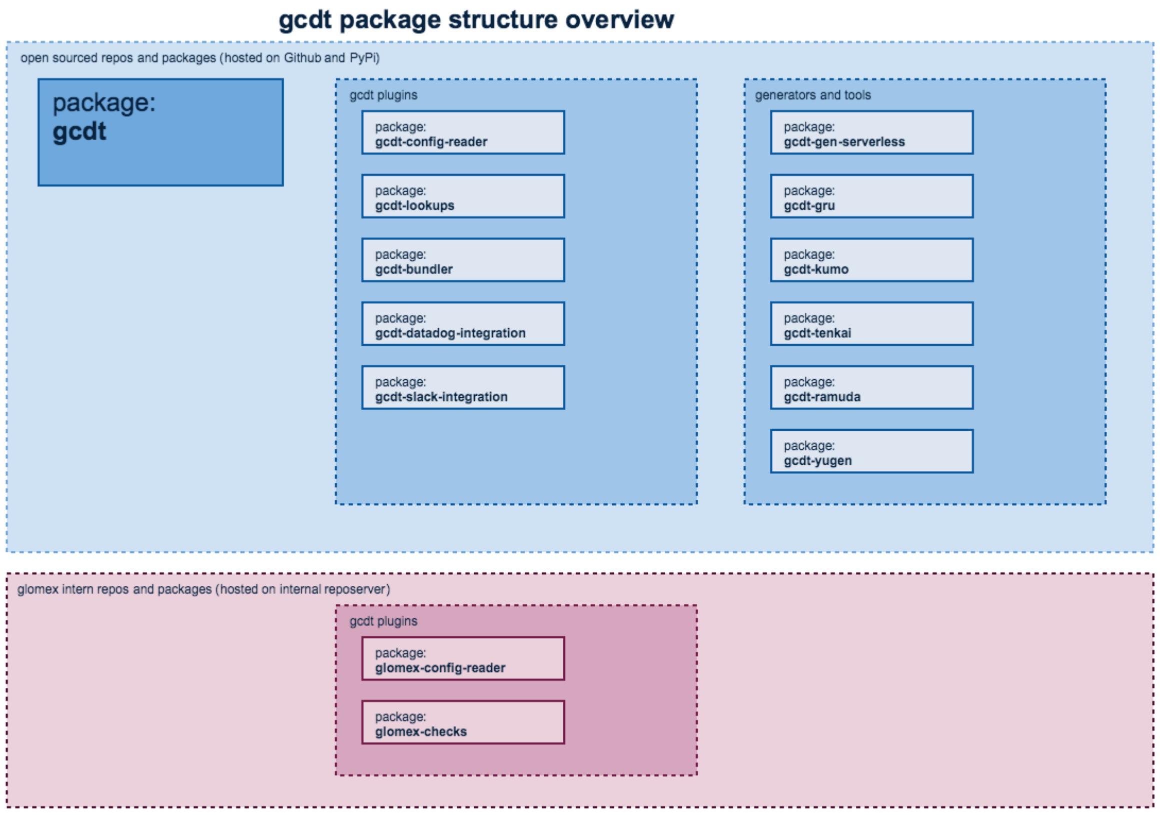
1. **Always use `virtualenv`.** A virtualenv is effectively an overlay on top of your system Python install. Creating a virtualenv can be thought of as copying your system Python environment into a local location. When you modify virtualenvs, you are modifying an isolated container. Modifying virtualenvs has no impact on your system Python.
2. **Use `pip` for installing packages.** Python packaging has historically been a mess. There are a handful of tools and APIs for installing Python packages. As a casual Python user, you only need to know of one of them: pip. If someone says install a package, you should be thinking create a `virtualenv`, activate a `virtualenv`, `pip install <package>`. You should never run `pip install` outside of a `virtualenv`. (The exception is to install `virtualenv` and pip itself, which you almost certainly want in your system/global Python.)
3. **Use `requirements` file for installing all project dependencies**. Always strictly specify the package version. Bad one: `somepackage=>2.0.3`. Good one: `somepackage==2.0.3`

Here is some useful links if you want dive deeper into Python package management.

4.3 gcdt package structure

The following diagram gives an overview on the gcdt packages. Please note how we grouped the gcdt packages in the following categories:

- gcdt - the gcdt core (livecycle mechanism, gcdt tools)
- gcdt plugins - packages to customize how you use gcdt
- gcdt generators and tools - scaffolding and tools to make your work even more efficient


gcdt package structure overview
open sourced repos and packages (hosted on Github and PyPi)
package: gcdt
gcdt plugins
package: gcdt-config-reader
package: gcdt-lookups
package: gcdt-bundler
package: gcdt-datadog-integration
package: gcdt-slack-integration
generators and tools
package: gcdt-gen-serverless
package: gcdt-gru
package: gcdt-kumo
package: gcdt-tenkai
package: gcdt-ramuda
package: gcdt-yugen
glomex intern repos and packages (hosted on internal reposerver)
gcdt plugins
package: glomex-config-reader
package: glomex-checks

gcdt package structure overview

At glomex we have very few (currently one) gcdt packages we do not want to open-source. The glomex-config-reader has very opinionated defaults on how we use gcdt on our AWS infrastructure that is very specific and optimized for our media usecase.

4.4 Maintaining dependencies for your project

It is a very common practice not to install Python packages by hand. Instead dependencies and version are managed in a documented and repeatable way. Basically you add the names and versions of your packages to a text file. Most projects also group their dependencies into `direct` dependencies of the service or application and packages they need to develop, build, test and document.

The grouping is not enforced by packaging but to have a std. within an organization is beneficial especially if your want to reuse CI/CD tools.

A little opinionated but pretty common:

- `requirements.txt` tools and packages your service directly depends on
- `requirements_def.txt` tools and packages you need to develop and test your service
- `requirements_gcdt.txt` gcdt and gcdt plugins you use to deploy your service to AWS

The easiest way to install gcdt is via pip and virtualenv.

4.5 Defining which gcdt-plugins to use

gcdt needs at least some gcdt-glugins so you should want to install these together. Add `gcdt` and the plugins you use to *requirements_gcdt.txt*

```
gcdt
gcdt-say-hello
gcdt-config-reader
gcdt-lookups
gcdt-bundler
gcdt-slack-integration
gcdt-datadog-integration
gcdt-gen-serverless
gcdt-kumo
gcdt-tenkai
gcdt-ramuda
gcdt-yugen
```

This is also a best practice to use the same `requirements_gcdt.txt` file on your build server, too.

4.6 Setup virtualenv

Using virtualenvs for Python is considered best practice. This is what you need to do:

- create a virtualenv (‘$ virtualenv venv’)
- install the packages you want to use (see above)
- a virtualenv works basically like every other technical device, you need to switch it on before you can use it (‘$ source ./venv/bin/activate’)

Prepare the venv:

```
$ virtualenv venv
```

Activate the venv for use:

```
$ source ./venv/bin/activate
```

4.7 Installing all dev dependencies in one go

Install the dependencies into venv:

```
$ pip install -U -r requirements_gcdt.txt
```

Now you can start using gcdt:

```
$ gcdt version
```

BTW, `gcdt version` shows you all the versions of gcdt and installed plugins. So you can use this to quickly check which plugins are installed.

4.8 Deactivate a virtualenv

I do not throw away my lawn mower once I am done but with my terminals I do that. But you can deactivate a virtualenv:

```
$ deactivate
```

4.9 Updating gcdt

You should frequently update your gcdt installation to get access to new features and bugfixes. When updating your gcdt installation, please update gcdt and all the plugins. Just updating gcdt or a single plugin could easily break your gcdt installation.

```
$ pip install -U -r requirements_gcdt.txt
```

This will update `gcdt` and all `gcdt plugins` specified in *requirements_gcdt.txt*

4.9.1 General remarks on "breaking changes" and deprecated features

We have two conflicting goals when maintaining gcdt:

- we want to introduce new features and replace older ones with newer implementations
- we want to be compatible with existing infrastructure and configurations

Besides that this is a pretty standard situation for a deployment tool and many other software projects. Nevertheless we want to make it explicit and consequently document here how we handle this.

We make sure that gcdt does work with our existing configurations. Frequent releases are ok but our squads expect downward compatibility for new patch versions. We could accept minor releases with breaking changes but expect an "update-documentation" which documents all steps that are necessary to upgrade.

We recently discussed and confirmed this in the gcdt grooming on 05.07.17 with representatives of the CO and VE squads.

In case we need to deprecate something we announce that in the gcdt changelog respectively in the changelog of gcdt plugins and tools.

The following sections go into `dependency specification` and provide information for how to upgrade from one version to the next.

4.9.2 Dependency specification (pinning versions)

Like we said above we guarantee compatibility with your installation procedures and configurations for patch versions. If you need to enforce compatibility it is recommended to best pin gcdt packages to minor versions in `requirements_gcdt.txt`.

This is just a sample, you need to pin to the actual versions you want to use:

```
gcdt==0.1.*
gcdt-config-reader==0.0.*
gcdt-lookups==0.0.*
gcdt-bundler==0.0.*
gcdt-slack-integration==0.0.*
gcdt-datadog-integration==0.0.*
gcdt-kumo==0.2.*
gcdt-tenkai==0.2.*
gcdt-ramuda==0.2.*
gcdt-yugen==0.2.*
```

Detailed information on Version Identification and Dependency Specification.

4.9.3 Updating gcdt from 0.1.x to 0.2.x

v 0.2.x removed long deprecated hook mechanism

Removed long deprecated hook support from kumo, tenkai and ramuda. No more 'pre_bundle', 'pre_hook', 'pre_create_hook', 'pre_update_hook', 'post_create_hook', 'post_update_hook', 'post_hook' any more.

Please use the gcdt lifecycle hook mechanism instead:

kumo lifecycle hooks using hooks in gcdt list of available hooks to use

with v 0.2.x you explicitly need to install all used gcdt tools

You need to install gcdt-tools (gcdt-kumo, gcdt-tenkai, gcdt-ramuda, gcdt-yugen). Please add the tools to your `requirements_gcdt.txt` like described in the installation section above.

with v 0.2.x we remove the deprecated ami lookup

Newer glomex `base_ami` uses a different naming scheme. Consequently the ami lookup implemented in gcdt provides you with old base_ami ids. Current working mode is to put the ami id into credstash (key: 'ops.dev.base_ami').

with v 0.2.x we moved cloudformation helpers to gcdt_kumo

If you use `iam` and `route53` helpers you need to change imports for these submodules from `gcdt` to `gcdt_kumo`.

with v 0.2.x we introduce config validation for all gcdt tools and plugins

The new config validation looks for required properties and does format checks in some cases. gcdt now also validates datatypes. We found some errors where `string type` was uses in existing configurations instead of the correct `integer type` and you need to fix this before your configuration can pass as valid:

A sample for a valid use of integer values:

```
"lambda": {
  ...
  "memorySize": 128,
  "timeout": 15,
}
```

4.9.4 Updating gcdt from 0.0.x to 0.1.x

Initially gcdt was a “monolithic” application. We wanted to a plugin mechanism that gives us a little bit more flexibility so we can install `generators` (scaffolding), `plugins`, and `tools` as needed. With 0.1.x versions you need to maintain a `requirements_gcdt.txt` file for you project in order to define which plugins and tools you want to use (see installation section above).

CHAPTER 5

gcdt command

glomex-cloud-deployment-tools

5.1 Related documents

5.1.1 Setting the ENV variable

You you need to set an environment variable “ENV” which indicates the account/staging area you want to work with. This parameter tells the tools which config file to use. For example if you want to set the environment variable ENV to ‘DEV’ you can do that as follows:

```
export ENV=DEV
```

5.2 Usage

To see available commands, call gcdt without any arguments:

```
$ gcdt
Usage:
        gcdt config
        gcdt version
```

5.3 Commands

5.3.1 config

This command is intended to provide tooling support for maintaining configuration.

The `config` command does the following:

- read configuration defaults
- read the config from file ('gcdt_.json')
- run the lookups
- format and output the config to the console

5.3.2 version

If you need help please ask on the gcdt slack channel or open a ticket. For this it is always great if you are able to provide information about the gcdt version you are using. A convenient way to find out the version of your gcdt install provides the following command:

```
$ gcdt version
WARNING: Please consider an update to gcdt version: 0.1.433
gcdt version 0.1.432
gcdt plugins:
 * gcdt-config-reader version 0.0.11
 * gcdt-bundler version 0.0.27
 * gcdt-slack-integration version 0.0.11
 * gcdt-datadog-integration version 0.0.15
 * gcdt-lookups version 0.0.12
```

`gcdt version` also provides you with an easy way to check whether a new release of gcdt is available.

CHAPTER 6

kumo command

`kumo` (from Japanese: cloud) is gcdts cloudformation deploy tool.

6.1 Related documents

6.2 Usage

To see available commands, call kumo without any arguments:

```
Usage:
        kumo deploy [--override-stack-policy] [-v]
        kumo list [-v]
        kumo delete -f [-v]
        kumo generate [-v]
        kumo preview [-v]
        kumo dot [-v]
        kumo stop [-v]
        kumo start [-v]
        kumo version

-h --help           show this
-v --verbose        show debug messages
```

6.3 Commands

6.3.1 deploy

will create or update a CloudFormation stack

to be able to update a stack that is protected by a stack policy you need to supply "–override-stack-policy"

6.3.2 list

will list all available CloudFormation stacks

6.3.3 delete

will delete a CloudFormation stack

6.3.4 generate

will generate the CloudFormation template for the given stack and write it to your current working directory.

6.3.5 preview

will create a CloudFormation ChangeSet with your current changes to the template

6.3.6 dot

Visualize the cloudformation template of your stack using `kumo dot`.

Sample Cloudformation

Installation of the dot binary is required on your Mac to convert the graph into svg (http://www.graphviz.org/Download_macos.php).

```
$ brew install graphviz
```

6.3.7 stop

```
"kumo stop" is a brand new feature we start rolling out to glomex AWS accounts.
We would like your feedback. Please talk to us if you require any improvements
(additional resources etc.).
```

Use `kumo stop` to stop resources contained in your cloudformation stack using `kumo stop`.

`kumo stop` currently comprises of the following features:

- resize autoscaling group to minSize=0, maxSize=0
- stop Ec2 instances
- stop RDS

Add this optional configuration to the `kumo` section of the config file to exclude your stack resources from start / stop.

```
...
"deployment": {
  "DisableStop": true
}
```

6.3.8 start

```
"kumo start" is a brand new feature we start rolling out to glomex AWS accounts.
We would like your feedback. Please talk to us if you require any improvements.
```

Start resources contained in your cloudformation stack using `kumo start`.

`kumo start` currently comprises of the following features:

- start RDS
- start EC2 instances
- restore autoscaling group minSize, maxSize to original values

6.3.9 version

will print the version of gcdt you are using

6.4 Folder Layout

The folder layout looks like this:

`cloudformation.py` -> creates troposphere template, needs a method like this:

```
def generate_template():
    return t.to_json()
```

`gcdt_dev.json` -> settings for dev in json format, needs to include all parameters for the cloudformation template + stack name

Further settings files, depending on your environments in the format of `gcdt_<ENV>.json`

6.4.1 Config file example

```
"stack": {
    "StackName": "sample-stack"
}
```

You like examples better than documentation? Check out our sample-stack at https://github.com/glomex/gcdt-sample-stack/tree/master/infrastructure

6.4.2 Configuring RoleARN for a cloudformation stack

There is a new Feature in CloudFormation which lets a User specify a Role which shall be used to execute the Stack. Docs can be found at http://docs.aws.amazon.com/AWSCloudFormation/latest/APIReference/API_CreateStack.html This can be used to limit access of users drastically and only give CloudFormation the permission to do all the heavy lifting.

```
"stack": {
    "RoleARN": "arn:aws:iam::<AccountID>:role/<CloudFormationRoleName>"
}
```

Make sure the role may be assumed by CloudFormation. See also: http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/using-iam-servicerole.html

6.4.3 Configuring NotificationARNs for a cloudformation stack

Amazon Simple Notification Service topic Amazon Resource Names (ARNs) that AWS CloudFormation associates with the stack.

```
{
  "kumo": {
    "stack": {
      "StackName": "infra-dev-kumo-sample-stack",
      "NotificationARNs": [
        "arn:aws:sns:eu-west-1:123456789012:mytopic1",
        "arn:aws:sns:eu-west-1:123456789012:mytopic2"
      ]
    },
    ...
```

Specify an empty list to remove all notification topics.

6.4.4 Setting the ENV variable

You you need to set an environment variable "ENV" which indicates the account/staging area you want to work with. This parameter tells the tools which config file to use. For example if you want to set the environment variable ENV to 'DEV' you can do that as follows:

```
export ENV=DEV
```

This can also be exploited to have different configuration for different regions which is not yet directly supported.

```
export ENV=DEV_eu-west-1
```

Will load the config file named `gcdt_dev_eu-west-1.json`

6.5 Howto

1. create and fill `cloudformation.py` with the contents of your stack
2. create and fill `settings_<env>.conf` with valid parameters for your CloudFormation template
3. call `kumo deploy` to deploy your stack to AWS

6.6 Kumo lifecycle hooks

Kumo lifecycle hooks work exactly like gcdt lifecycle hooks but have a specialized integration for kumo templates.

```
def my_hook(params):
    context, config = params
    ...
```

(continues on next page)

(continued from previous page)

```
def register():
    """Please be very specific about when your hook needs to run and why.
    E.g. run the sample stuff after at the very beginning of the lifecycle
    """
    gcdt_signals.initialized.connect(my_hook)


def deregister():
    gcdt_signals.initialized.disconnect(my_hook)
```

One kumo speciality for the `command_finalized` hook is that you can access the context attribute 'stack_output'. to access and use outputs of your stack within the hook implementation.

6.7 DEPRECATED Kumo legacy hooks

The hooks in this section are deprecated please use gcdt lifecycle hooks (see above).

Please note the legacy hooks will be removed with the next minor release (v 0.2.0).

kumo offers numerous hook functions that get called during the lifecycle of a kumo deploy run:

- pre_hook()
  - gets called before everything else - even config reading. Useful for e.g. creating secrets in credstash if they don't exist
- pre_create_hook()
  - gets called before a stack is created
- pre_update_hook()
  - gets called before a stack is updated
- post_create_hook()
  - gets called after a stack is created
- post_update_hook()
  - gets called after a stack is updated
- post_hook()
  - gets called after a stack is either updated or created

You can basically call any custom code you want. Just implement the function in `cloudformation.py`

Multiple ways of using parameters in your hook functions:

- no arguments (as previous to version 0.0.73.dev0.dev0)
- use kwargs dict and just access the arguments you need e.g. "def pre_hook(**kwargs):"
- use all positional arguments e.g. "def pre_hook(awsclient, config, parameters, stack_outputs, stack_state):"
- use all arguments as keyword arguments or mix.
- with version 0.0.77 we decided to move away from using boto_sessions towards awsclient (more flexible and low-level).

6.8 Using gcdt functionality in your cloudformation templates

Historically `cloudformation.py` templates imported functionality from gcdt and glomex_utils packages. With version 0.0.77 we consolidated and copied `get_env` over to gcdt.utils.

Made functionality available in gcdt (sounds awful but it was there already anyway) :

- gcdt.utils: get_env now available

Continued no changes:

- gcdt.iam: IAMRoleAndPolicies

The following functionality requires `awsclient` to lookup information from AWS. The `awsclient` is available in the cloudformation template only within the scope of a hook (see above). Consequently you need to execute your calls within the scope of a hook:

- gcdt.servicediscovery: get_outputs_for_stack
- gcdt.route53: create_record
- gcdt.kumo_util: ensure_ebs_volume_tags_autoscaling_group

6.9 Accessing context and config in cloudformation

In the last few month we learned about a few usecases where it is desired to have access to config and context within your template. We had some workarounds using hooks but now there is a proper implementation for this feature.

In order to access context and config in your `cloudformation.py` you need to add both `context` and `config` as arguments to the 'generate_template? function of your template:

```python
def generate_template(context, config):
    template = troposphere.Template()
    ph.initialize(template, 'miaImportProcessor')
    assemble_particles(template, context, config)
    return template.to_json()
```

In case you do not want to use this information in your template you don't have to use it (like before).

```python
def generate_template():
    ...
```

6.10 Stack Policies

kumo does offer support for stack policies. It has a default stack policy that will get applied to each stack:

```json
{
  "Statement" : [
    {
      "Effect" : "Allow",
      "Action" : "Update:Modify",
      "Principal": "*",
      "Resource" : "*"
    },
    {
```

(continues on next page)

(continued from previous page)

```
      "Effect" : "Deny",
      "Action" : ["Update:Replace", "Update:Delete"],
      "Principal": "*",
      "Resource" : "*"
    }
  ]
}
```

This allows an update operation to modify each resource but disables replacement or deletion. If you supply "–override-stack-policy" to kumo then it will use another default policy that gets applied during updates and allows every operation on every resource:

```
{
  "Statement" : [
    {
      "Effect" : "Deny",
      "Action" : "Update:*",
      "Principal": "*",
      "Resource" : "*"
    }
  ]
}
```

If you want to lock down your stack even more you can implement two functions in your cloudformation.py file:

- get_stack_policy()
- – the actual stack policy for your stack
- get_stack_policy_during_update()
- – the policy that gets applied during updates

These should return a valid stack policy document which is then preferred over the default value.

6.11 Signal handling

kumo receives a SIGINT or SIGTERM signal during a stack update `cancel_update_stack` is called for the stack.

CHAPTER 7

kumo particles

At glomex we create our infrastructure via AWS cloudformation and in order to do that efficiently we wanted to use reusable building blocks. For that to achieve we where evaluating different solutions available to us via open source.

So this is basically conceptual paper-ware (structure and a few helpers). You still need to write the particles.

Please be aware not to let kumo particles stop you from anything. In case you do not have particles or you do not want to write any you can still build beautiful infrastructure from the raw services provided by AWS.

kumo particles are perfectly optional. There is no tight coupling! You can totally bring your own building-block-mechanism and still use kumo for deployment. You do not even have to use troposphere - as long as your mechanism can export a valid json cloudformation template we are fine. Actually we encourage you to do so. Please share with us what you come up with.

7.1 Related documents

7.2 Goals

- codify best practices for infrastructure
- use `cloudformation.py` to assemble a stack from particles
- complexity is handled in particle

7.3 Detailed requirements

- particle has default parameters that can be overridden
- particle provides default permission that can be overridden
- we distribute particles as python packages (later move away from github subprojects)
- we want to distribute generic std. particles company wide (e.g. glomex-particles)

- we want to distribute squad specific particles (e.g. mes-particles)

7.4 Status on kumo particles implementation

- kumo particle implementation is based on MES `template_generator.py`
- answered "what is the minimum information we need to provide to use a particle?"
- restore troposphere character (talked about context => template is the context)
- added SERVICE_NAME and DEFAULT_TAGS to template
- I liked the "template_generator" implementation but class structure gets in the way when creating stacks from multiple particle sources
- move cloudformation parameters and outputs into particle
- move permissions profile to particle

TODOs

- create particle profiles using awacs to further shrink the particles
- look into naming conventions / tooling for autogenerated resource names here it is important that in case we generate random names we can regenerate the same names during testing (gcdt placebo tools)
- share particles via package (need Github repo, Jenkins build, . . .)

7.5 Usage

To build better infrastructure at glomex we want to assemble infrastructure from reusable particles.

The `gcdt.kumo_particle_helper` module contains the functionality (initialize, get_particle_permissions, and Particle) to integrate the kumo particles into troposphere

7.6 Sample particles

7.6.1 instance

will create or update a CloudFormation stack

7.7 Quickstart example using particles

With kumo particles you can import particles from multiple sources:

```
from gcdt_kumo import kumo_particle_helper as ph
import eventbus_particle as eb
import reusable_particles as rp
```

We use `cloudformation.py` to assemble a stack from particles:

```
def assemble_particles(template):
    ################# parameters #################################################
    param_sns_alerts_topic = template.add_parameter(troposphere.Parameter(
        'SNSAlertsTopic',
        Description='Name for topic that receive notifications for validation.',
        Type='String'
    ))

    ################# particles ##################################################
    particles = []  # list of Particle()
    sg_frontend_web = Ref('%sFrontendWeb' % template.SERVICE_NAME)

    ################# s3 bucket ##################################################
    particles.append(rp.create_s3_bucket(template))
    ...
```

Under the hood we use `troposphere` to code cloudformation templates. The troposphere template instance is used as a common `context` to exchange information between kumo particles. With kumo each `cloudformation.py` needs to implement a generate_template function.

```
def generate_template():
    template = troposphere.Template()
    ph.initialize(template, 'miaImportProcessor')
    assemble_particles(template)
    return template.to_json()
```

7.8 Developing your own particles

We just started with kumo particles and plan to provide more help on particle development in the future.

CHAPTER 8

tenkai command

`tenkai` (from Japanese: deployment) is gcdts codedeploy tool.

8.1 Related documents

8.2 Usage

To see available commands, call this:

```
Usage:
        tenkai bundle [-v]
        tenkai deploy [-v]
        tenkai version

-h --help           show this
-v --verbose        show debug messages
```

8.2.1 deploy

bundles your code then uploads it to S3 as a new revision and triggers a new deployment

8.2.2 version

will print the version of gcdt you are using

8.3 Folder Layout

codedeploy -> folder containing your deployment bundle codedeploy_env.conf -> settings for your code

```
"codedeploy": {
    "applicationName": "mep-dev-cms-stack2-mediaExchangeCms-F5PZ6BM2TI8",
    "deploymentGroupName": "mep-dev-cms-stack2-mediaExchangeCmsDg-1S2MHZ0NEB5MN",
    "deploymentConfigName": "CodeDeployDefaultemplate.AllAtOnce01",
    "artifactsBucket": "7finity-portal-dev-deployment"
}
```

8.4 tenkai configuration

8.4.1 add stack_output.yml to your tenkai bundle

If you need a convenient way of using the stack output during codedeploy on your instance then you can use this feature.

`tenkai` adds a `stack_output.yml` to the bundle artifact if you add the following configuration:

```
{
    'stack_output': 'lookup:stack:<your_stack_name>'
    ...
}
```

8.4.2 Adding a settings.json file

tenkai supports a `settings` section. If it is used a `settings.json` file is added to the zip bundle containing the values. You can specify the settings within the `tenkai` section.

```
    ...
    "settings": {
        "MYVALUE": "FOO"
    }
```

You can use lookups like for the rest of the configuration. Note that the values are looked up BEFORE the the instance is deployed via codedeploy. If values change during the instance lifecycle it does not recognise the changes. For values that must be updated you should lookup the values in your code using for example credstash.

```
    ...
    "settings": {
        "accountId": "lookup:stack:infra-dev:AWSAccountId"
    }
```

8.4.3 Configure log group

In case `tenkai deploy` fails we attempt to provide the log output from the ec2 instance to ease your troubleshooting. The default log group is '/var/log/messages'. In case your ec2 instances are configured to log into another log group you can provide the necessary log group configuration to tenkai like this:

```
"tenkai": {
    ...
    "deployment": {
```

(continues on next page)

(continued from previous page)

```
        "LogGroup": "/my/loggroup"
    }
}
```

Note: as a convention each ec2 instances has its own log stream with using the instanceId as name of the stream.

8.4.4 Setting the ENV variable

You you need to set an environment variable "ENV" which indicates the account/staging area you want to work with. This parameter tells the tools which config file to use. For example if you want to set the environment variable ENV to 'DEV' you can do that as follows:

```
export ENV=DEV
```

8.5 Signal handling

tenkai receives a SIGINT or SIGTERM signal during a deployment `stop_deployment` is called for the running deployment with `autoRollbackEnabled`.

CHAPTER 9

ramuda command

`ramuda` (from Japanese: lambda) is gcdts AWS lambda deployment tool.

9.1 Related documents

9.2 Usage

To see available commands, call this:

```
Usage:
        ramuda clean
        ramuda bundle [--keep] [-v]
        ramuda deploy [--keep] [-v]
        ramuda list
        ramuda metrics <lambda>
        ramuda info
        ramuda wire [-v]
        ramuda unwire [-v]
        ramuda delete [-v] -f <lambda> [--delete-logs]
        ramuda rollback [-v] <lambda> [<version>]
        ramuda ping [-v] <lambda> [<version>]
        ramuda invoke [-v] <lambda> [<version>] [--invocation-type=<type>] --payload=
↪<payload> [--outfile=<file>]
        ramuda logs <lambda> [--start=<start>] [--end=<end>] [--tail]
        ramuda version

Options:
-h --help                show this
-v --verbose             show debug messages
--keep                   keep (reuse) installed packages
--payload=payload        '{"foo": "bar"}' or file://input.txt
--invocation-type=type   Event, RequestResponse or DryRun
```

(continues on next page)

(continued from previous page)

```
--outfile=file          write the response to file
--delete-logs           delete the log group and contained logs
--start=start           log start UTC '2017-06-28 14:23' or '1h', '3d', '5w', ...
--end=end               log end UTC '2017-06-28 14:25' or '2h', '4d', '6w', ...
--tail                  continuously output logs (can't use '--end'), stop 'Ctrl-C'
```

9.2.1 clean

removes local bundle files.

9.2.2 bundle

zips all the files belonging to your lambda according to your config and requirements.txt and puts it in your current working directory as `bundle.zip`. Useful for debugging as you can still provide different environments.

9.2.3 deploy

Deploy an AWS Lambda function to AWS. If the lambda function is non-existent it will create a new one.

For an existing lambda function ramuda checks whether the hashcode of the bundle has changed and updates the lambda function accordingly. This feature was added to ramuda so we are able to compare the hashcodes locally and save time for bundle uploads to AWS.

This only works if subsequent deployments are executed from the same virtualenv (and same machine). The current implementation of the gcdt-bundler starts every deployment with a fresh virtualenv. If you want the hashcode comparison you need to provide the `--keep` option. With the '–keep' option the virtualenv is preserved. Otherwise the hashcodes of the ramuda code bundles will be different and the code will be deployed.

If you can not reuse ('–keep') the virtualenv for example in case you deploy from different machines you need to use `git` to check for code changes and skip deployments accordingly.

In any case configuration will be updated and an alias called "ACTIVE" will be set to this version.

9.2.4 list

lists all existing lambda functions including additional information like config and active version:

```
dp-dev-store-redshift-create-cdn-tables
        Memory: 128
        Timeout: 180
        Role: arn:aws:iam::644239850139:role/lambda/dp-dev-store-redshift-cdn-
↪LambdaCdnRedshiftTableCr-G7ME657RXFDB
        Current Version: $LATEST
        Last Modified: 2016-04-26T18:03:44.705+0000
        CodeSha256: KY0Xk+g/Gt69V0siRhgaG7zWbg234dmb2hoz0NHIa3A=
```

9.2.5 metrics

displays metric for a given lambda:

```
dp-dev-ingest-lambda-cdnnorm
        Duration 488872443
        Errors 642
        Invocations 5202
        Throttles 13
```

9.2.6 wire

"wires" the AWS Lambda function to an event source.

9.2.7 unwire

delets the event configuration for the lambda function

9.2.8 delete

deletes a lambda function

If you use the `--delete-logs` the cloudwatch log group associated to the AWS Lambda function is deleted including log entries, too. This helps to save cost for items used in testing.

9.2.9 rollback

sets the active version to ACTIVE -1 or to a given version

9.2.10 invoke

In this section, you invoke your Lambda function manually using the `ramuda invoke` command.

```
$ ramuda invoke my_hello_world \
--invocation-type RequestResponse \
--payload '{"key1":"value1", "key2":"value2", "key3":"value3"}'
```

If you want you can save the payload to a file (for example, input.txt) and provide the file name as a parameter:

```
$ ramuda invoke my_hello_world \
--invocation-type RequestResponse \
--payload file://input.txt
```

The preceding invoke command specifies RequestResponse as the invocation type, which returns a response immediately in response to the function execution. Alternatively, you can specify Event as the invocation type to invoke the function asynchronously.

9.2.11 logs

The `ramuda logs` command provides you with convenient access to log events emitted by your AWS Lambda function.

The command offers '–start' and '–end' options where you can filter the log events to your specification. You can use human readable dates like '2017-07-24 14:00:00' or you can specify dates in the past relative to `now` using '1m', '2h', '3d', '5w', etc.

```
$ ramuda logs ops-dev-captain-crunch-slack-notifier --start=1d
MoinMoin fin0007m!
2017-07-07
07:00:32  START RequestId: f75cd7de-62e1-11e7-937d-ef5726c6f5c8 Version: $LATEST
07:00:36  END RequestId: f75cd7de-62e1-11e7-937d-ef5726c6f5c8
07:00:36  REPORT RequestId: f75cd7de-62e1-11e7-937d-ef5726c6f5c8        Duration:␣
↪4221.50 ms    Billed Duration: 4300 ms        Memory Size: 128 MB     Max Memory␣
↪Used: 43 MB
Bye fin0007m. Talk to you soon!
```

The '–start' option has a default of '1d'. This means if you run `ramuda logs <your-function-name>` you get the log output of your function for the last 24 hours.

```
$ ramuda logs ops-dev-captain-crunch-slack-notifier
MoinMoin fin0007m!
2017-07-07
07:00:32  START RequestId: f75cd7de-62e1-11e7-937d-ef5726c6f5c8 Version: $LATEST
07:00:36  END RequestId: f75cd7de-62e1-11e7-937d-ef5726c6f5c8
07:00:36  REPORT RequestId: f75cd7de-62e1-11e7-937d-ef5726c6f5c8        Duration:␣
↪4221.50 ms    Billed Duration: 4300 ms        Memory Size: 128 MB     Max Memory␣
↪Used: 43 MB
Bye fin0007m. Talk to you soon!
```

You can use `ramuda logs` to **tail** the log output of your lambda function. The default start date in tail mode is 5 minutes before. You can specify any past start date in tail mode but you can not specify an '–end' option in tail mode. To exit the `ramuda logs` tail mode use `Ctrl-C`.

```
$ ramuda logs ops-dev-captain-crunch-slack-notifier --start=1d --tail
MoinMoin fin0007m!
Use 'Ctrl-C' to exit tail mode
2017-07-07
07:00:32  START RequestId: f75cd7de-62e1-11e7-937d-ef5726c6f5c8 Version: $LATEST
07:00:36  END RequestId: f75cd7de-62e1-11e7-937d-ef5726c6f5c8
07:00:36  REPORT RequestId: f75cd7de-62e1-11e7-937d-ef5726c6f5c8        Duration:␣
↪4221.50 ms    Billed Duration: 4300 ms        Memory Size: 128 MB     Max Memory␣
↪Used: 43 MB
^CReceived SIGINT signal - exiting command 'ramuda logs'
```

9.2.12 version

will print the version of gcdt you are using

9.3 Folder Layout

9.4 Sample config file

sample gcdt_dev.json file:

```
{
  "ramuda": {
    "lambda": {
      "name": "dp-dev-store-redshift-load",
      "description": "Lambda function which loads normalized files into redshift",
      "role": "arn:aws:iam::644239850139:role/lambda/dp-dev-store-redshift-cdn-lo-
↪LambdaCdnRedshiftLoad-DD2S84CZFGT4",
      "handlerFunction": "handler.lambda_handler",
      "handlerFile": "handler.py",
      "timeout": "180",
      "memorySize": "128",
      "events": [
        {
          "event_source": {
            "arn":  "arn:aws:s3:::my-bucket",
            "events": ["s3:ObjectCreated:*"]
          }
        },
        {
          "event_source": {
            "name": "send_reminder_to_slack",
            "schedule": "rate(1 minute)"
          }
        }
      ],
      "vpc": {
        "subnetIds": [
          "subnet-87685dde",
          "subnet-9f39ccfb",
          "subnet-166d7061"
        ],
        "securityGroups": [
          "sg-ae6850ca"
        ]
      }
    },
    "bundling": {
      "zip": "bundle.zip",
      "preBundle": [
        "../bin/first_script.sh",
        "../bin/second_script.sh"
      ],
      "folders": [
        {
          "source": "../redshiftcdnloader",
          "target": "./redshiftcdnloader"
        },
        {
          "source": "psycopg2-linux",
          "target": "psycopg2"
        }
      ]
    },
    "deployment": {
      "region": "eu-west-1",
      "artifactBucket": "7finity-$PROJECT-deployment"
    }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

9.5 ramuda configuration as part of the gcdt_.json file

9.5.1 log retention

Possible values for the log retention in days are: 1, 3, 5, 7, 14, 30, 60, 90, 120, 150, 180, 365, 400, 545, 731, 1827, and 3653.

```
{
    "lambda": {
        ...
        "logs": {
            "retentionInDays": 90
        }
}
```

9.5.2 S3 upload

ramuda can upload your lambda functions to S3 instead of inline through the API. To enable this feature add this to the "ramuda" section of your `gcdt_<env>.json` config file:

```
"deployment": {
    "region": "eu-west-1",
    "artifactBucket": "7finity-$PROJECT-deployment"
}
```

You can get the name of the bucket from Ops and it should be part of the stack outputs of the base stack in your account (s3DeploymentBucket).

9.5.3 runtime support

gcdt supports the `nodejs4.3`, `nodejs6.10`, `python2.7`, `python3.6` runtimes.

Add the runtime config to the `lambda` section of your gcdt configuration.

```
    "runtime": "nodejs4.3"
```

At this point the following features are implemented:

- install dependencies before bundling (dependencies are defined in package.json)
- bundling (bundle the lambda function code and dependencies)
- deployment (the nodejs4.3 lambda function is setup with the nodejs4.3 runtime)
- configuration (bundles `settings_<env>.conf` file for your environments)
- nodejs support is tested by our automated gcdt testsuite
- if no runtime is defined gcdt uses the default runtime `python2.7`

Note: for this to work you need to **have npm installed** on the machine you want to run the ramuda bundling!

9.5.4 AWS Lambda environment variables

Ramuda supports AWS Lambda environment variables. You can specify them within the `lambda` section.

```
    ...
    "environment": {
        "MYVALUE": "FOO"
    }
```

More information you can find in AWS docs.

9.5.5 Adding a settings.json file

Ramuda supports a settings section. If used a `settings.json` file is added to the zip bundle. You can specify the settings within the `ramuda` section.

```
    ...
    "settings": {
        "MYVALUE": "FOO"
    }
```

You can use lookups like for the rest of the configuration. Note that the values are looked up BEFORE the AWS Lambda function is deployed. If values change during the AWS Lambda function lifecycle it does not recognise the changes. For values that must be updated you should lookup the values in your code using for example credstash.

```
    ...
    "settings": {
        "accountId": "lookup:stack:infra-dev:AWSAccountId"
    }
```

9.5.6 Adding event configuration

gcdt can be used to easily schedule functions to occur on regular intervals. Just list your expressions to schedule them using cron or rate syntax in your gcdt_.json config file like this:

```
...
"events": [{
    "event_source": {
        "name": "send_reminder_to_slack",
        "schedule": "rate(1 minute)"
    }
}]
```

The schedule expression defines when to execute your lambda function in cron or rate format.

Supported event types.

Similarly, you can have your functions execute in response to events that happen in the AWS ecosystem, such as S3 uploads, Kinesis streams, and SNS messages, etc..

In your gcdt_.json config file, define your event sources. The following sample config will execute your AWS Lambda function in response to new objects in your my-bucket S3 bucket. Note that your function must accept event and context parameters.

```
...
"events": [{
    "event_source": {
        "arn":  "arn:aws:s3:::my-bucket",
        "events": ["s3:ObjectCreated:*"],
        "suffix": ".jpg"
    }
}],
```

Similarly, for a Simple Notification Service (SNS) event:

```
...
"events": [{
    "event_source": {
        "arn":  "arn:aws:sns:::your-event-topic-arn",
        "events": ["sns:Publish"]
    }
}]
```

Kinesis is slightly different as it is not event-based but pulling from a stream:

```
...
"events": [{
    "event_source": {
        "arn": arn:aws:kinesis:eu-west-1:1234554:stream/your_stream"
        "starting_position": "TRIM_HORIZON",
        "batch_size": 50,
        "enabled": true
    }
}]
```

Lambda@Edge needs a CloudFront event trigger:

```
...
"events": [{
    "event_source": {
        "arn": "arn:aws:cloudfront::420189626185:distribution/E1V934UN4EJGJA",
        "cache_behavior": "*",
        "cloudfront_event": "origin-request"
    }
}]
```

Sample CloudWatch event pattern (parameter store change):

```
  ...
  "events": [
    {
      "event_source": {
        "name": "ssm_parameter_changed",
        "input_path": "$.detail",
        "pattern": {
          "source": [
            "aws.ssm"
          ],
          "detail-type": [
            "Parameter Store Change"
          ]
```

(continues on next page)

(continued from previous page)

```
        }
      }
    }
  ],
```

Sample CloudWatch event pattern (ECS container instance state change):

```
...
"events": [
  {
    "event_source": {
      "name": "ecs_container_instance_state_change",
      "pattern": {
        "source": [
          "aws.ecs"
        ],
        "detail-type": [
          "ECS Container Instance State Change"
        ]
      }
    }
  }
],
```

The request for AWS Lambda triggers for SQS came up a few times. This is NOT directly supported by AWS since SQS was released ~10 years earlier than AWS Lambda. But there are a different workarounds available - choose whatever is most appropriate for your situation:

If you don't need near real-time processing, two valid options are:

- Create CloudWatch Event Rule that will trigger the Lambda function every N minutes (e.g. every minute).
- Create CloudWatch alarm watching ApproximateNumberOfMessagesVisible parameter for your SQS queue. This alarm should publish to an SNS topic, which in turn will trigger Lambda function.

Possible real-time solutions:

- Replace SQS queue with Kinesis or DynamoDB. Both can trigger Lambda functions on updates.
- Inject SNS before SQS. SNS can add items to SQS and trigger Lambda function.

more details on SQS triggers

9.6 Deploying AWS Lambda@Edge

AWS Lambda@Edge is relatively new so we have to deal with some (hopefully temporary) limitations (like it is only available in Virginia):

- can not use artifact bucket for upload
- max memory use 128 MB
- max timeout 3 seconds for 'origin request' and 'origin response' events; for 'viewer request' and 'viewer response' events timeout is 1 second
- lambda@edge does not implement ALIAS or $LATEST so we use the version-nbr of the last published version of the lambda function for wiring

- unwire removes the lambda trigger for the configured CloudFront distribution (regardless if the lambda function is the same as the configured one or not)
- 'ramuda wire' and 'ramuda unwire' finish after initiation of the replication to CloudFront. This means CloudFront distribution is in state "Pending" when ramuda exits.
- you cannot delete lambda functions that have been replicated to CloudFront edge locations

9.7 Setting the ENV variable

You you need to set an environment variable "ENV" which indicates the account/staging area you want to work with. This parameter tells the tools which config file to use. For example if you want to set the environment variable ENV to 'DEV' you can do that as follows:

```
export ENV=DEV
```

9.8 Environment specific configuration for your lambda functions

Please put the environment specific configuration for your lambda function into a `gcdt_<env>.json` file. For most teams a useful convention would be to maintain at least 'dev', 'qa', and 'prod' envs.

9.9 Defining dependencies for your NodeJs lambda function

A sample `package.json` file to that defines a dependency to the `1337` npm module:

```
{
  "name": "my-sample-lambda",
  "version": "0.0.1",
  "description": "A very simple lambda function",
  "main": "index.js",
  "dependencies": {
    "1337": "^1.0.0"
  }
}
```

9.10 Sample NodeJs lambda function

From using lambda extensively we find it a good practise to implement the `ping` feature. With the ping `ramdua` automatically checks if your code is running fine on AWS.

Please consider to implement a `ping` in your own lambda functions:

```
var l33t = require('1337')


exports.handler = function(event, context, callback) {
    console.log( "event", event );

    if (typeof(event.ramuda_action) !== "undefined" && event.ramuda_action == "ping")
↪{
```

(continues on next page)

(continued from previous page)

```
        console.log("respond to ping event");
        callback(null, "alive");
    } else {
        console.log(l33t('glomex rocks!'));  // 910m3x r0ck5!
        callback();  // success
    }
};
```

CHAPTER 10

yugen command

`yugen` (from Japanese: “dim”, “deep” or “mysterious”) is gcdts API Gateway deployment tool.

10.1 Related documents

10.2 Usage

To see available commands, call this:

```
Usage:
        yugen deploy [-v]
        yugen delete -f [-v]
        yugen export [-v]
        yugen list [-v]
        yugen apikey-create <keyname> [-v]
        yugen apikey-list [-v]
        yugen apikey-delete [-v]
        yugen custom-domain-create [-v]
        yugen version

-h --help           show this
-v --verbose        show debug messages
```

10.2.1 deploy

creates/updates an API from a given swagger file

10.2.2 export

exports the API definition to a swagger file

10.2.3 list

lists all existing APIs

10.2.4 apikey-create

creates an API key

10.2.5 apikey-list

lists all existing API keys

10.2.6 apikey-delete

deletes an API key

10.2.7 version

will print the version of gcdt you are using

10.3 Folder Layout

`swagger.yaml` -> API definition in swagger with API Gateway extensions

```
{
    "yugen": {
        "api": {
            "name": "dp-dev-serve-api-2",
            "description": "description",
            "targetStage": "dev",
            "apiKey": "xxx",
            "cacheClusterEnabled": true
            "cacheClusterSize": "0.5"
            "methodSettings": {
                "/path/to/resource/GET": {
                    "cachingEnabled": false
                }
            }
        }
    },
    "ramuda": {
        "lambda": {

            "entries": [
              {
                "name": "dp-dev-serve-api-query",
                "alias": "ACTIVE"
              },
              {
                "name": "dp-dev-serve-api-query-elasticsearch",
```

(continues on next page)

(continued from previous page)

```
                "alias": "ACTIVE"
              }
            ]
        ...
        }
    }
}
```

Set the config attribute `cacheClusterEnabled` to `true` in your gcdt_.json config file to enable a cache cluster for the specified stage resource.

Set the config attribute `cacheClusterSize` to '0.5'|'1.6'|'6.1'|'13.5'|'28.4'|'58.2'|'118'|'237' in your gcdt_.json config file to configure the size for an enabled cache cluster. Default setting is '0.5'.

The config attribute `methodSettings` allows you to define settings related to a setting_key. A `setting_key` is defined as <resource_path>/<http_method>. So it is important that your setting_key contains the http_method (GET, PUT, OPTIONS, etc.), too. You can specify method setting properties as defined in the AWS docs: https://botocore.readthedocs.io/en/latest/reference/services/apigateway.html#APIGateway.Client.update_stage like for example 'cachingEnabled', 'loggingLevel', etc.

10.3.1 Create custom domain

Currently the certificates need to be deployed in `us-east-1` and used in the `certificateArn` in the `customDomain` section. If you use ACM lookup (gcdt-lookups) to lookup your certificate arn for yugen it uses `us-east-1` already.

```
"customDomain": {
  "basePath": "",
  "certificateName": "wildcard.glomex.com-2017-3-2",
  "certificateArn": "lookup:acm:*.infra.glomex.cloud",
  "domainName": "unittest-gcdt-sample-api-dev-eu-west-1.dev.mes.glomex.cloud",
  "hostedDomainZoneId": "lookup:stack:infra-dev:internalDomainHostedZoneID",
  "route53Record": "unittest-gcdt-sample-api-dev-eu-west-1.dev.infra.glomex.cloud",
  "ensureCname": true
}
```

10.3.2 Setting the ENV variable

You you need to set an environment variable "ENV" which indicates the account/staging area you want to work with. This parameter tells the tools which config file to use. For example if you want to set the environment variable ENV to 'DEV' you can do that as follows:

```
export ENV=DEV
```

CHAPTER 11

Plugins for gcdt

11.1 Introduction

Beginning with version 0.0.77 gcdt supports plugins. Plugins are a way to add features to gcdt without having to directly modify the gcdt core.

This gcdt plugin userguide aims to make it easy for you to get started using plugins in your projects. gcdt-plugins help you customize the gcdt tools towards your specific project needs.

gcdt plugins are available for many different areas from reading your specific configuration format, to looking up credentials from your secret-store. gcdt plugins were implemented internally at glomex.

glomex – The Global Media Exchange – is a provider of a global, open marketplace for premium video content as well as a technical service provider for the entire value chain of online video marketing.

gcdt plugins and userguide are released under MIT License.

The gcdt plugins userguide starts with this introduction, then provides an overview on how to use and configure gcdt plugins in general. The following parts each cover one gcdt plugin.

This user guide assumes that you know gcdt and the AWS services you want to automate so we do not cover AWS services in great detail and instead point to relevant documentation. But even if you are starting out on AWS, gcdt will help you to quickly leave the AWS webconsole behind and to move towards infrastructure-as-code.

11.1.1 Related documents

This section aims to provide to you a list of related documents that will be useful to gain a detailed understanding about what the gcdt tool suite does. With this background you will be able to tap into the full potential of the gcdt tools.

11.2 Overview

This sections provides and overview on the gcdt-plugin system. It covers everything necessary to understand how it works at a high level.

11.2.1 Plugin system key functions

Main functionality of the gcdt plugin system is to provide means so gcdt can be customized and extended without changing the core functionality. This is necessary for example in case not everybody uses slack or datadog. In this situation one can just not use the plugin or use a plugin which supports an alternate service.

Plugins are added to gcdt via python packages. The following section covers how to do that.

The gcdt plugin mechanism also encapsulates the plugin code in a way that it is separated from gcdt core. This enables us to change and test a plugin as a component independent from gcdt core. More details about the `plugin mechanism` are covered in the next chapter.

11.2.2 Plugin installation

Plugins are maintained as standard python packages. Just install plugins you want via `pip install <plugin_name>`. Same idea applies to removing plugins from a project setup. Using `pip uninstall <plugin_name>` removes the plugin.

A more sophisticated way of doing that which goes well with CI/CD is to simply add your gcdt plugins to your projects `requirements_gcdt.txt` file. Especially if you need more tools and plugins this makes setting up your CI environment easy and reproducible. `pip install -r requirements.txt -r requirments_dev.txt` installs all the packages you need for your service and for developing it.

11.2.3 Plugin configuration

Configuration for a plugin are specific for that plugin so please consult the plugins documentation for specific configuration options. General mechanism is that you add the configuration for the plugin to your `gcdt_<env>.json` file. Add a section with the plugin name like in the following snippet:

```
    ...
    'plugins': {
        ...
        'gcdt_slack_integration': {
            'slack_webhook': 'lookup:secret:slack.webhook:CONTINUE_IF_NOT_FOUND'
        },
        ...
    }
```

11.2.4 Plugin descriptions

The following table lists the plugins and gives a brief overview what each plugin is used for.

Plugin | Description —— | ——— gcdt_config_reader | read configuration files in json, python, or yaml format glomex_config_reader | read hocon configuration files gcdt_lookup | lookup information related to your AWS account gcdt_bundler | create code bundles for tenkai and ramuda. gcdt_say_hello | simple plugin to demonstrate how plugins work / are developed gcdt_slack_integration | send deployment status information to slack gcdt_datadog_integration | send deployment metrics and events to datadog

Please refer to detailed plugin's documentation later in this document folder for detailed information about that plugin.

Later we are going to put plugins in separate repositories so they can have independent owners and development / release cycles. With that we move the detailed plugin documentation to the plugin README and documentation.

11.3 gcdt plugin mechanism

The previous chapter gave an overview on the gcdt plugin system so please make sure that you have read through that one. This section goes into more detail in how the gcdt plugin mechanism works. So you can customize plugins or even write new ones.

gcdt plugins are standard python packages which are installed separately. How to do this is covered in the previous chapter plugin overview. To understand how the plugin mechanism works one must know about the `gcdt lifecycle` which is covered in the next section.

If a gcdt command is entered on the command line things are processed in the following order:

- Python interpreter loads the gcdt package
- CLI options and arguments are parsed
- we check if any relevant plugins are installed (details in `plugin entry_points`)
- relevant plugins are loaded and check if they comply to gcdt-plugin structure.
- register() function of each plugin is called.
- then the gcdt lifecycle is executed
- each lifecycle step fires an event which we call `signal`

11.3.1 Anatomy of a plugin

Each gcdt-plugin must implement `register()` and `deregister()` functions to be a valid gcdt-plugin that can be used. Note how the `register()` function connects the plugin function `say_hello` with the `initialized` lifecycle step. `deregister()` just disconnects the plugin functionality from the gcdt lifecycle.

```
def say_hello(context):
    """
    :param context: The boto_session, etc.. say_hello plugin needs the 'user'
    """
    print('MoinMoin %s!' % context.get('user', 'to you'))

...

def register():
    """Please be very specific about when your plugin needs to run and why.
    E.g. run the sample stuff after at the very beginning of the lifecycle
    """
    gcdt_signals.initialized.connect(say_hello)
    gcdt_signals.finalized.connect(say_bye)


def deregister():
    gcdt_signals.initialized.disconnect(say_hello)
    gcdt_signals.finalized.disconnect(say_bye)
```

Handing in information to your plugin functions. Look into gcdt.gcdt_signals for details.

```
def my_plug_function(params):
    """
    :param params: context, config (context - the env, user, _awsclient, etc..
                   config - The stack details, etc..)
    """
    context, config = params
    # implementation here
    ...
```

All gcdt lifecycle steps provide the `(context, config)` tuple besides `initialized` and `finalized`. These two only provide the context.

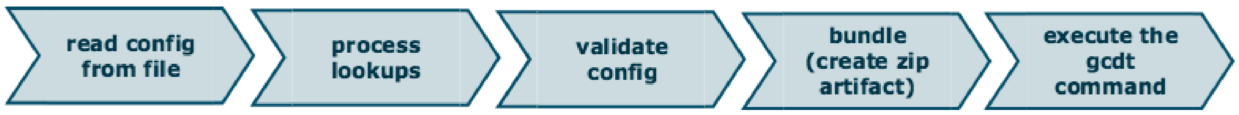
The same lifecycle & signals mechanism applies to gcdt hooks. So if you ever wondered how gcdt hooks are work - now you know.

11.3.2 Overview of the gcdt lifecycle

The gcdt lifecycle is the essential piece of the gcdt tool core. It is like the clockwork of a watch. The gcdt lifecycle makes sure that everything is executed in the right order and everything works together like commands, hooks, plugins, etc.

The gcdt lifecycle is generic. This means the gcdt lifecycle is the same for each and every gcdt tool. But it is possible that a tool does not need a certain lifecycle step to it just skips it. For example there is no bundling for kumo(, yet?).

The coarse grained gcdt lifecycle looks like that:



gcdt_lifecycle

If during processing of a lifecycle an error occurs then the processing stops.

11.3.3 List of gcdt signals

The list of gcdt signals you can use in plugins or hooks:

- initialized - after reading arguments and context
- config_read_init
- config_read_finalized
- check_credentials_init
- check_credentials_finalized
- lookup_init
- lookup_finalized
- config_validation_init
- config_validation_finalized
- bundle_pre - we need this signal to implement the prebundle-hook
- bundle_init
- bundle_finalized

- command_init
- command_finalized
- error
- finalized

The order of this list also represents the order of the lifecycle steps with in the gcdt lifecycle.

11.3.4 Developing plugins

If you want to develop a plugin to integrate some service or to optimize the configuration for your environment we recommend that you "fork" the say_hello plugin so you have the right structure and start from there.

If you need help developing your plugin or want to discuss your plans and get some feedback please don't be shy. The SRE squad is here to help.

11.3.5 Testing a plugin

Testing a gcdt plugin should be easy since its code is decoupled from gcdt core. It is a good practice to put the tests into the `tests` folder. Also please prefix all your test files with `test_` in this way pytest can pick them up. Please make sure that your plugin test coverage is on the save side of 80%.

11.4 Overview on configuration

Configuration and configuration file formats are very dear to us and we already know to you, too. For example the data_platform squad likes to use multiple tool specific hocon files so this functionality is provided by the 'glomex_config_reader?. Other squads like to have all the configuration for a service environment in one single config file (gcdt_config_reader).

Regarding configuration file formats many things are possible and it should be very simple to implement your specific requirements as a plugin so you could use XML or INI format. Talk to us and do not shy away from good old json config.

11.4.1 Structure of the configuration (internal representation)

The datastructure gcdt uses to internally represent the configuration basically is json compatible.

We have configuration on the following levels:

- top-level
- tool-level (tools are kumo, tenkai, ramuda, yugen)
- plugin specific configuration

```
{
    'kumo': {
        ...
    },
    'tenkai': {
        ...
    },
    'ramuda': {
```

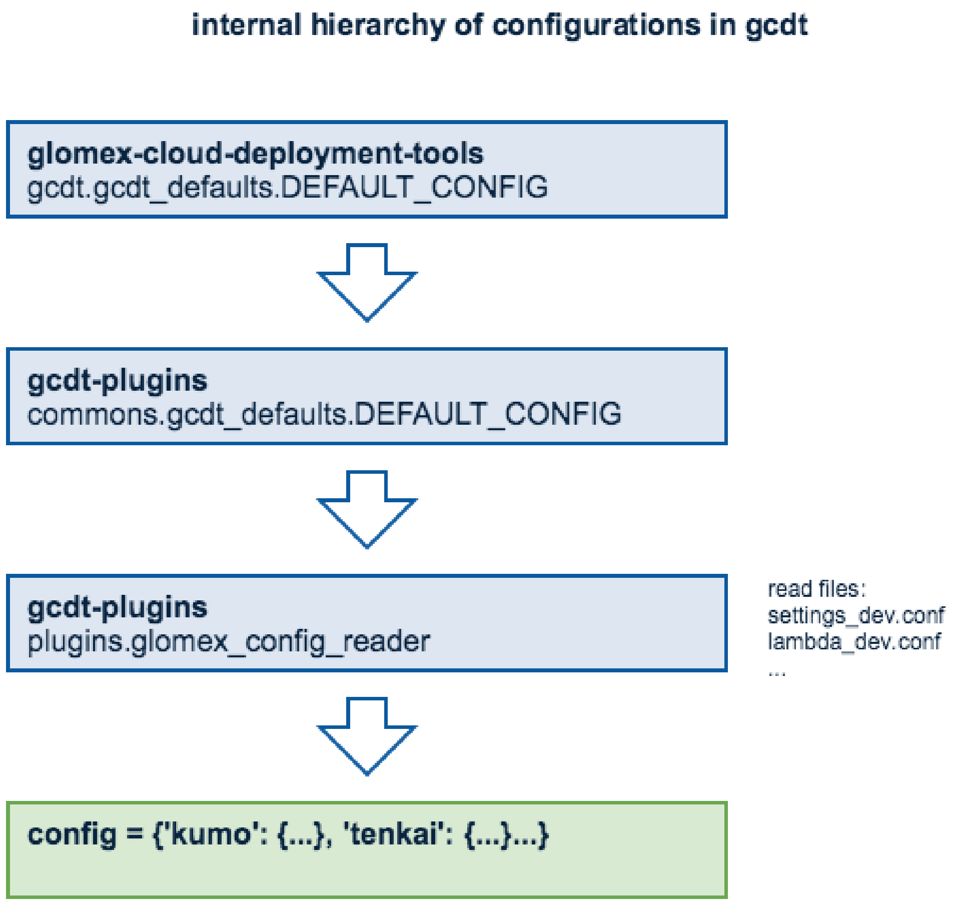
(continues on next page)

(continued from previous page)

```
        ...
    },
    'yugen': {
        ...
    },
    'plugins' {
        'plugin_a': {
            ...
        },
        ...
    }
}
```

11.4.2 Multiple levels of gcdt configuration

Configuration is assembled in multiple levels:


internal hierarchy of configurations in gcdt
glomex-cloud-deployment-tools
gcdt.gcdt_defaults.DEFAULT_CONFIG
gcdt-plugins
commons.gcdt_defaults.DEFAULT_CONFIG
gcdt-plugins
plugins.glomex_config_reader
read files:
settings_dev.conf
lambda_dev.conf
...
config = {'kumo': {...}, 'tenkai': {...}...}

gcdt configuration defaults

The multiple levels of configurations represent different stages in the lifecycle process. This allows to have a very generic "catch-all" configuration but to override this configuration in specific cases when we have more specific information. Like when using a plugin. For example the glomex_config_reader works on hocon files so it looks for files with a `.conf` extension whereas the gcdt_config_reader looks for `.json` config files.

11.4.3 Context

The gcdt `context` is the "internal" datastructure gcdt is using to process the CLI command that need to be executed. So for instance each plugin or hook can find out about the `tool`, `command`, or `env` it is currently processing. With the context we follow the convention to prefix private attributes with an '_' like with the _`awsclient` that plugins use to access AWS services but _`awsclient` does not show up in the slack or datadog notification.

```
{
  'version': '0.1.426',
  'command': 'preview',
  '_awsclient': <gcdt.gcdt_awsclient.AWSClient object at 0x10291a490>,
  'env': 'dev',
  '_arguments': {
    '--override-stack-policy': False,
    '-f': False,
    'delete': False,
    'deploy': False,
    'dot': False,
    'generate': False,
    'list': False,
    'preview': True,
    'version': False
  },
  'tool': 'kumo',
  'plugins': [
    'gcdt-config-reader',
    'gcdt-bundler',
    'gcdt-say-hello',
    'glomex-config-reader',
    'gcdt-slack-integration',
    'gcdt-gru',
    'gcdt-datadog-integration',
    'gcdt-lookups'
  ],
  'user': 'fin0007m'
}
```

11.5 gcdt-config-reader plugin

Read config from files in json, python or yaml format. There is a section called `overview on configuration` above. Please make sure you have that one covered.

11.5.1 Related documents

11.5.2 json configuration files

The gcdt_config_reader plugin allows us to have configurations in json format.

The configuration files are environment specific. This means the config file looks like gcdt_.json' where stands for the environment you use (some thing like dev, stage, prod, etc.).

11.5.3 yaml configuration files

The gcdt_config_reader plugin allows us to have configurations in `yaml` format.

The configuration files are environment specific. This means the config file looks like gcdt_.yaml‘ where stands for the environment you use (some thing like dev, stage, prod, etc.).

11.5.4 python configuration files

The gcdt_config_reader plugin allows us to have configurations in python format (with a .py extension).

The configuration files are environment specific. This means the config file looks like gcdt_.py‘ where stands for the environment you use (some thing like dev, stage, prod, etc.).

The python configuration files have a few specifics which are documented here.

The python configuration only work if they follow the convention to implement a `generate_config()` function. The `generate_config()` needs to return a dictionary with the configuration. Please follow the configuration structure described below.

```
def generate_config():
    return CFG
```

You can also use hooks in the python config files. Just implement the `register`, `deregister` like they are described in the plugin section to make that work.

11.5.5 gcdtignore patterns

gcdt supports multiple ways of ignoring files from bundling. The file format of pattern files is the same as gitignore files. This means you can use wildcards to exclude files from bundling like for example `*.pyc`. This is currently relevant for the gcdt tools `tenkai` and `ramuda`.

The following files are supported:

- .ramudaignore - in the user home directory
- .gcdtignore - in the current directory
- .npmignore - in the current directory

Alternatively you can provide the ignore pattern with your `gcdt_<env>.json` config file:

```
{
    "gcdtignore": ["*.pyc", "trash"]
    ...
}
```

On a more technical note: all ignore patterns you provide are consolidated by the config reader and provided to plugins and tools as `config['gcdtignore']`.

11.5.6 reference to base config file

The gcdt_config_reader plugin supports a `baseconfig` property which gives a basepath. This works with all supported config file formats like json, yaml and .py config files.

```
{
  "baseconfig": "baseconfig",
  "ramuda": {
    "lambda": {
      "runtime": "nodejs4.3",
  ...
```

With that you can for example implement the following config structure with your config files:

```
baseconfig.json
-> gcdt_dev.json
-> gcdt_stage.json
```

In this sample `baseconfig.json` contains all common config values for all environments. `gcdt_dev.json` contains only values specific for the development environment (same with `gcdt_stage.json`). Config values can be overridden by the dependent `gcdt_<env>.json` file, too.

11.6 gcdt-lookups plugin

The lookups functionality is pinned to a dedicated gcdt lifecycle step.

11.6.1 Related documents

11.6.2 lookup stack output

The `stack` lookup is used to substitute configuration where the value is an output from another cloudformation stack.

format:	*lookup:stack:<stackname>:<output>*
sample:	*lookup:secret:slack.webhook*

regional lookup of stack output:

format:	*lookup:region:<region>:stack:<stackname>:<output>*
sample:	*lookup:region:us-east-1:secret:slack.webhook*

11.6.3 DEPRECATED lookup ssl certificate

format:	*lookup:ssl:<stackname>:<output>*
sample:	*lookup:ssl:*.infra.glomex.cloud*

‘ssl’ lookup uses the `server_certificate` functionality built into AWS IAM. It is configured default lookup so for each stack also the certificates are added to stackdata.

This is DEPRECATED! If possible, please use the acm lookup!

11.6.4 lookup acm certificate

format:	*lookup:acm:<name_1>:...:<name_n>:*
sample:	*lookup:acm:foo.mes.glomex.cloud:supercars.infra.glomex.cloud:*.dev.infra.glomex.cloud*

'acm' lookup uses the AWS ACM (Certificate Manager) functionality. It is configured as default lookup.

Features of the `acm lookup`:

- pass a list of hostnames that should be secured.
- check all certificates in ACM if the configured CN (DomainName) or SANs (SubjectAlternativeNames) (including wildcards) if they match for the given list of hostnames
- the chosen certificates STATUS must be **ISSUED**
- if there are multiple matches, use the one with the most distant expiry date
- return the ARN of the certificate
- wildcards for hosted zone are expressed with "*."
- 'ERROR' in case a certificate matching the specified list of names can not be found

Note: if you use ACM lookup in yugen / API Gateway you need to deploy the certificates to the `us-east-1` region.

11.6.5 lookup secret

The `secret` lookup is used to substitute configuration where the value is a password, token or other sensitive information that you can not commit to a repository.

lookup the 'datadog_api_key' entry from credstash:

format:	*lookup:secret:<name>.<subname>*
sample:	*lookup:secret:datadog.api_key*

regional lookup of secret:

format:	*lookup:region:<region>:secret:<name>.<subname>*
sample:	*lookup:region:us-east-1:secret:datadog.api_key*

lookup the 'slack.webhook' entry from credstash:

sample:	*lookup:secret:slack.webhook:CONTINUE_IF_NOT_FOUND*

note that the `slack.webhook` lookup does not fail it the accounts credstash does not have the `slack.token` entry.

11.6.6 DEPRECATED lookup baseami

The `baseami` lookup is used lookup the baseami for cloudformation infrastructures.

11.7 gcdt-bundler plugin

Create code bundles for tenkai and ramuda.

11.7.1 Related documents

11.7.2 Sample config

bundling sample for kumo:

```
...
"bundling": {
    "folders": [
        { "source" = "./codedeploy", target = "." },
        { "source" = "../app", target = "app" },
        { "source" = "../images", target = "images" },
        { "source" = "../supercars", target = "supercars" }
    ]
}
```

Bundling sample for ramuda:

```
...
"bundling": {
  "zip": "bundle.zip",
  "folders": [
    { "source": "./vendored", "target": "."},
    { "source": "./impl", "target": "impl"}
  ]
},
```

11.8 gcdt-say-hello plugin

This is probably the friendliest plugin ever. It simply greets the user.

Purpose of this plugin is to demonstrate how the gcdt plugin mechanism works to developers. You can use it as blueprint to jump-start developing your own plugin.

```python
# -*- coding: utf-8 -*-
"""A gcdt-plugin which demonstrates how to implement hello world as plugin."""
from __future__ import unicode_literals, print_function

from gcdt import gcdt_signals


def say_hello(context):
    """say hi.
    :param context: The boto_session, etc.. say_hello plugin needs the 'user'
    """
    print('MoinMoin %s!' % context.get('user', 'to you'))


def say_bye(context):
```

(continues on next page)

(continued from previous page)

```
    """say bye.
    :param context: The boto_session, etc.. say_hello plugin needs the 'user'
    """
    print('Bye %s. Talk to you soon!' % context.get('user', 'then'))


def register():
    """Please be very specific about when your plugin needs to run and why.
    E.g. run the sample stuff after at the very beginning of the lifecycle
    """
    gcdt_signals.initialized.connect(say_hello)
    gcdt_signals.finalized.connect(say_bye)


def deregister():
    gcdt_signals.initialized.disconnect(say_hello)
    gcdt_signals.finalized.disconnect(say_bye)
```

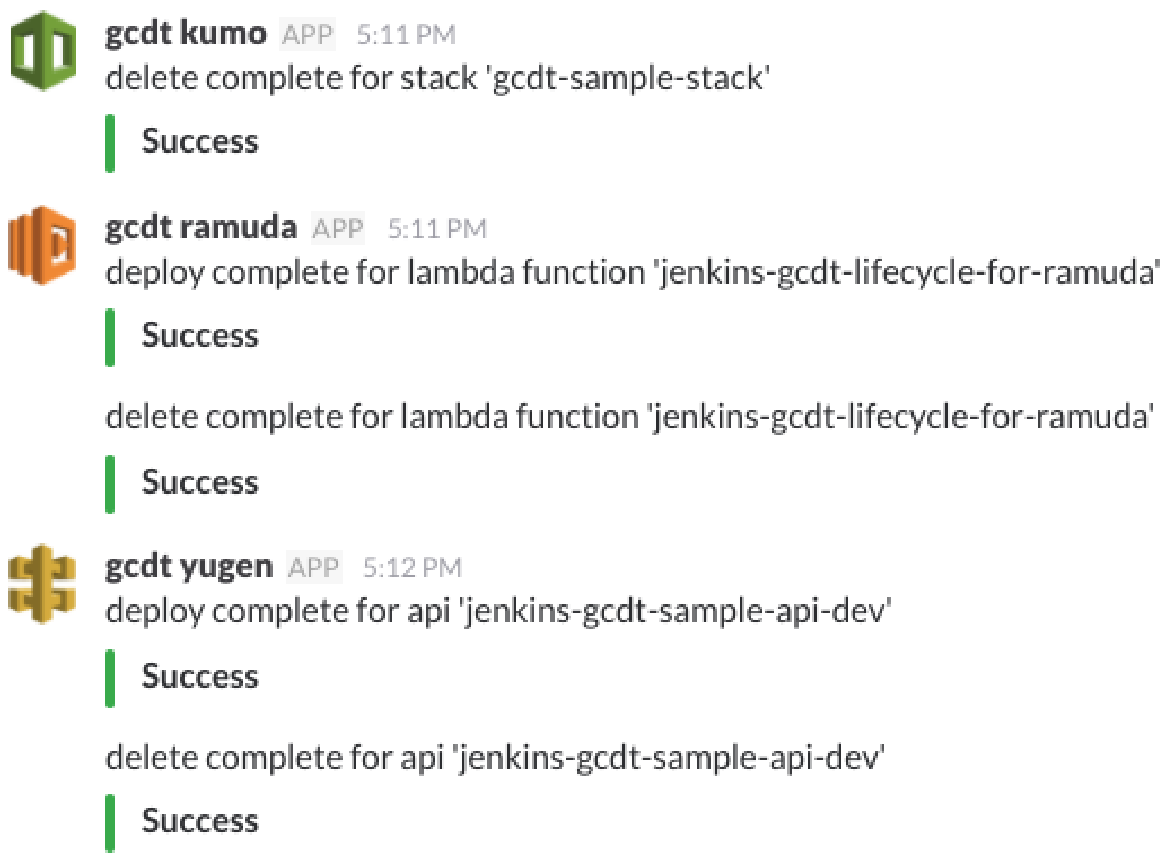
11.9 gcdt-slack-integration plugin

Announce the status of your deployments on slack.

11.9.1 Related documents

11.9.2 slack integration plugin functionality

Announce deployments on the slack channel for your squad:


gcdt kumo APP 5:11 PM
delete complete for stack 'gcdt-sample-stack'
Success

gcdt ramuda APP 5:11 PM
deploy complete for lambda function 'jenkins-gcdt-lifecycle-for-ramuda'
Success

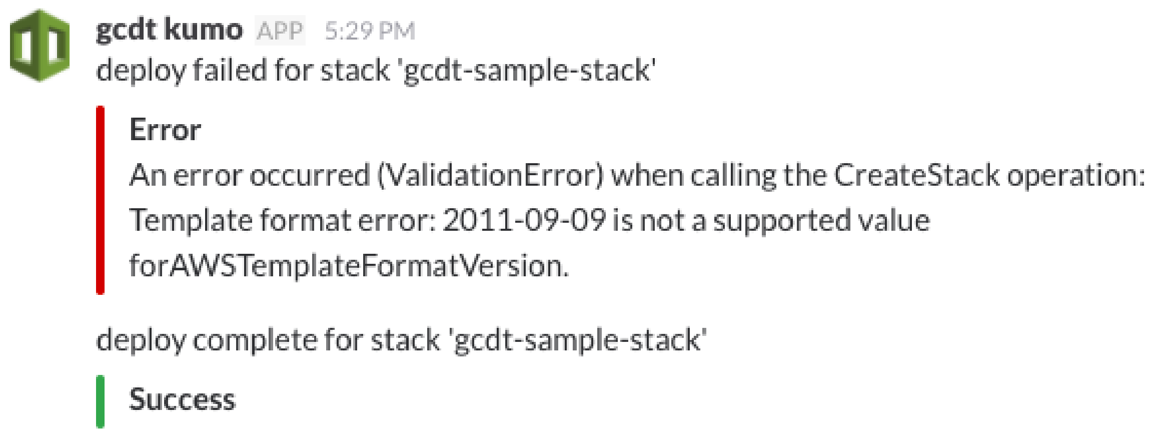
delete complete for lambda function 'jenkins-gcdt-lifecycle-for-ramuda'
Success

gcdt yugen APP 5:12 PM
deploy complete for api 'jenkins-gcdt-sample-api-dev'
Success

delete complete for api 'jenkins-gcdt-sample-api-dev'
Success

gcdt_slack integration

In case a deployments fails you get a notification, too:


gcdt kumo APP 5:29 PM
deploy failed for stack 'gcdt-sample-stack'
Error
An error occurred (ValidationError) when calling the CreateStack operation: Template format error: 2011-09-09 is not a supported value forAWSTemplateFormatVersion.

deploy complete for stack 'gcdt-sample-stack'
Success

gcdt_slack integration

11.9.3 Setup

To setup the slack integration for your account you need two things:

- a slack webhook
- you need to add the slack webhook to credstash so the `lookup:secret` works

Add a secret to credstash as follows (maybe check for an existing key first):

```
credstash put datadog_api_key <my_key>
```

11.9.4 Configuration

`datadog.api_key` is provided via secret lookup:

```
    ...
    'plugins': {
        'gcdt_slack_integration': {
            'channel': '<my_squad_slack_channel>'
        },
        ...
    }
```

Note the `slack_webhook` configuration is provided via default configuration. You do not need to change that as long as you are happy with the default config:

```
            'slack_webhook': 'lookup:secret:slack.webhook:CONTINUE_IF_NOT_FOUND'
```

11.10 gcdt-datadog-integration plugin

The original usecase for the gcdt_datadog_integration was to get usage data from gcdt executions. So we get some idea on frequency and gcdt utilization. We still use metrics with the datadog gcdt-dashboard to concentrate our efforts on the relevant parts of gcdt.

11.10.1 Related documents

11.10.2 datadog integration plugin functionality

- send metrics and events according to tool and command
- provide context e.g. host, user, env, stack

11.10.3 Configuration

`datadog.api_key` is provided via secret lookup:

```
'gcdt_datadog_integration': {
    'datadog_api_key': 'lookup:secret:datadog.api_key'
},
```

CHAPTER 12

Frequently Asked Questions (faq)

12.1 Homebrew Python

If you installed Python via Homebrew on OS X and get this error:

```
must supply either home or prefix/exec-prefix -- not both
```

You can find a solution on here

12.2 Python package errors

Please ensure that you have the latest version of `pip`, `setuptools` and `virtualenv`

If you have error like this:

```
pip._vendor.pkg_resources.DistributionNotFound:
```

or

```
pkg_resources.DistributionNotFound: regex==2017.6.07
```

you should update your `pip` and `virtualenv` packages

```
$ pip install -U pip
$ pip install -U virtualenv
```

12.3 Bundling error

This error is usually caused by not having installed the `gcdt-bundler` plugin:

```
(.python) root@:/app# AWS_PROFILE=superuser-dev ENV=qa ramuda deploy
ERROR: u'_zipfile'
ERROR: u'_zipfile'
Traceback (most recent call last):
  File "/root/.python/bin/ramuda", line 11, in <module>
    sys.exit(main())
  File "/root/.python/local/lib/python2.7/site-packages/gcdt/ramuda_main.py", line␣
↪255, in main
    dispatch_only=['version', 'clean']))
  File "/root/.python/local/lib/python2.7/site-packages/gcdt/gcdt_lifecycle.py", line␣
↪195, in main
    return lifecycle(awsclient, env, tool, command, arguments)
  File "/root/.python/local/lib/python2.7/site-packages/gcdt/gcdt_lifecycle.py", line␣
↪142, in lifecycle
    raise(e)
KeyError: u'_zipfile'
```

You need to add `gcdt-bundler` into *requirements_gcdt.txt* and do:

```
$ pip install -U -r requirements_gcdt.txt
```

12.4 Missing configuration error

After updating `gcdt` to the latest version you get the following error:

```
Configuration missing for ‘kumo’
```

This error appears if you used `hocon` based configs without having installed the `glomex-config-reader` plugin. You can install it or use conf2json util (only for glomex users) to transform your `hocon` configs into `json` one.

12.5 Environment variable error

If you run any `gcdt` commands (kumo, tenkai, ramuda etc) and get the following error:

```
ERROR: 'ENV' environment variable not set!
```

Environment variable "ENV" indicated the account/staging area you want to work with. This parameter tells the tools which config file to use. Please be sure that you provide the correct environment variables (ENV=PROD/DEV/etc.)

```
$ export ENV=DEV
```

12.6 Using hooks in gcdt

We implemented hooks in gcdt similar to the plugin mechanism.

You can use hooks in gcdt in the following places:

- use hooks in a `cloudformation.py` template
- use hooks in a `gcdt_<env>.py` config file
- use hooks in a `hookfile.py`. Please specify the location of the `hookfile` in your config file.

For details on gcdt_lifecycle and gcdt_signals please take a look into the gcdt-plugins section of this documentation.

CHAPTER 13

Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

13.1 [0.1.436] - 2017-08-17

13.1.1 Fixed

- yugen: default value for cache size property (#328)

13.2 [0.1.435] - 2017-08-17

13.2.1 Added

- yugen: add cache size property (#328)

13.3 [0.1.433] - 2017-08-14

13.3.1 Added

- kumo: stop / start cloudformation stack (#342)

13.4 [0.1.432] - 2017-08-10

13.4.1 Fixed

- gcdt: removed explicit requests dependency (#359)

13.5 [0.1.431] - 2017-08-10

13.5.1 Added

- gcdt-lookups: added acm lookup documentation, do acm lookup by default (#359)

13.5.2 Deprecated

- gcdt-lookups: ami was already deprecated but made it more explicit + docs

13.6 [0.1.430] - 2017-08-08

13.6.1 Added

- yugen: implement cache settings on method level (#328)

13.7 [0.1.429] - 2017-08-07

13.7.1 Added

- kumo: fixed compatibility issue with templates in update case (#357)

13.8 [0.1.428] - 2017-08-03

13.8.1 Added

- kumo: use ‘context’ and ‘config’ in cloudformation templates (#353)
- kumo: add ‘stack_output’ attribute to context (#353)
- added section on version pinning to installation guide

13.9 [0.1.427] - 2017-08-01

13.9.1 Added

- yugen: cacheClusterEnabled setting (#328)

13.10 [0.1.426] - 2017-08-01

13.10.1 Added

- kumo: kumo_particle_helper plus docs (#244)

13.11 [0.1.425] - 2017-08-01

13.11.1 Fixed

- kumo, tenkai: report correct deployment status on slack (#348)

13.12 [0.1.424] - 2017-08-01

13.12.1 Added

- ramuda: mechanism for flexible event wiring (#264)

13.12.2 Deprecated

- ramuda: ‘events’ config dictionary like ‘s3Sources’ and ‘timeSchedules’ (#264)

13.13 [0.1.423] - 2017-07-21

13.13.1 Added

- tenkai: improve output in case of errors (#316)

13.14 [0.1.422] - 2017-07-18

13.14.1 Fixed

- kumo: fix warning message for deprecated format (#337)

13.15 [0.1.421] - 2017-07-18

13.15.1 Fixed

- dependencies for logcapture mechanism (#285)

13.16 [0.1.420] - 2017-07-18

13.16.1 Added

- kumo: add SNS notifications (#185)
- logcapture mechanism for tests (#285)

13.16.2 Deprecated

- kumo: "cloudformation" config section, use "parameters" & "stack" instead (#337)

13.17 [0.1.419] - 2017-07-17

13.17.1 Added

- kumo: add status message for empty changeset (#126)

13.18 [0.1.418] - 2017-07-17

13.18.1 Added

- ramuda & tenkai settings files - json format (#295)

13.19 [0.1.417] - 2017-07-13

13.19.1 Added

- getting started guide (#312)

13.20 [0.1.415] - 2017-07-12

13.20.1 Fixed

- kumo update when using the artifactBucket setting (#332)

13.21 [0.1.413] - 2017-07-07

13.21.1 Added

- ramuda logs command (#247)

13.22 [0.1.412] - 2017-07-05

13.22.1 Added

- fix some docu issues for ramuda

13.23 [0.1.411] - 2017-07-04

13.23.1 Added

- configure cloudwatch logs for ramuda (#191)

13.24 [0.1.410] - 2017-07-03

13.24.1 Fixed

- fix ‘file://’ prefix for ramuda invoke payload (#246)

13.25 [0.1.409] - 2017-07-03

13.25.1 Added

- use roleARN for kumo delete, too (#162)

13.26 [0.1.408] - 2017-06-30

13.26.1 Added

- kumo preview for new stack (#73)

13.27 [0.1.407] - 2017-06-30

13.27.1 Fixed

- do not fail check_gcdt_update when PyPi is down (#313)

13.28 [0.1.406] - 2017-06-30

13.28.1 Added

- support for AWS Lambda ENV variables (#262)

13.29 [0.1.405] - 2017-06-30

13.29.1 Fixed

- minor documentation changes for kumo. Better description of usage of a role for CloudFormation (#162)

13.30 [0.1.404] - 2017-06-29

13.30.1 Added

- handle SIGTERM and SIGINT signals and stop running deployments accordingly (#40)

13.31 [0.1.403] - 2017-06-22

13.31.1 Added

- define GracefulExit exception (#40)

13.32 [0.1.402] - 2017-06-21

13.32.1 Fixed

- ramuda redeploy version without changes issue (#145)

13.33 [0.1.401] - 2017-06-20

13.33.1 Fixed

- kumo artifactBucket handling (#292)

13.34 [0.1.400] - 2017-06-20

13.34.1 Fixed

- datetime handling (#226)

13.35 [0.1.399] - 2017-06-19

13.35.1 Fixed

- tenkai clean up /tmp files (#60)

13.36 [0.1.398] - 2017-06-16

13.36.1 Added

- kumo use special role for cloudformation deployments (#162)

13.37 [0.1.397] - 2017-06-16

13.37.1 Fixed

- kumo parameter diffing without params fails (#64)
- kumo parameter diffing flaws (#184)

13.38 [0.1.396] - 2017-06-12

13.38.1 Added

- ramuda invoke command (#246)

13.39 [0.1.394] - 2017-06-09

13.39.1 Added

- add stack_output.yml to tenkai bundle artifact (#266)

13.40 [0.1.393] - 2017-06-09

13.40.1 Added

- ramuda works with python3.6

13.41 [0.1.392] - 2017-06-07

13.41.1 Added

- added some documentation related to gcdt-bundler and AWS Lambda runtimes

13.42 [0.1.391] - 2017-06-02

13.42.1 Added

- support for nodejs6.10 + python3.6 runtimes
- specify folders (source & target) for tenkai bundles
- ramuda '–keep' option to speed up your dev cycles

13.43 [0.1.390] - 2017-05-31

13.43.1 Added

- prepare for nodejs6.10 runtime (PR 293)

13.44 [0.1.8] - 2017-04-27

13.44.1 Added

- improve exception handling (#285) plus proper error message for missing config

13.45 [0.1.7] - 2017-04-27

13.45.1 Added

- check AWS Lambda runtime in ramuda config (#254)

13.46 [0.1.6] - 2017-04-26

13.46.1 Added

- getLogger helper to be used by gcdt-plugins (#213)

13.47 [0.1.5] - 2017-04-26

13.47.1 Added

- gcdt plugin version info in version cmd and datadog (#250)
- use gcdt plugins in E2E test lifecycle (#250)

13.48 [0.1.4] - 2017-04-07

13.48.1 Added

- gcdt package publicly available on PyPi (#250)

13.49 [0.1.0] - 2017-04-05

13.49.1 Added

- open source on Github (#255)
- moved build jobs to new infra jenkins (#255)

13.49.2 Changed

- it is now mandatory for gcdt users to maintain plugin dependencies

13.50 [0.0.84] - 2017-03-30

13.50.1 Added

- support for hooks in cloudformation templates and hookfiles (#218)

13.51 [0.0.83] - 2017-03-29

13.51.1 Added

- updated gcdt-plugin dependency

13.52 [0.0.82] - 2017-03-29

13.52.1 Added

- added scaffolding mechanism
- use MIT LICENSE (#253)

13.52.2 Fixed

- tamed greedy lookup (#258)
- kumo fixed deprecated pre_hook (#259)

13.53 [0.0.81] - 2017-03-24

13.53.1 Added

- included plugin documentation
- gcdt_config_reader for json config files (#218)

13.53.2 Fixed

- ramuda bundle includes settings_.conf file (#249)
- minor improvements from code review sessions (#230)
- missing environment is not properly handled (#248)

13.54 [0.0.80] - 2017-03-09

13.54.1 Fixed

- fixed ramuda vpc config handling (#225)

13.55 [0.0.79] - 2017-03-08

13.55.1 Fixed

- kumo docu bug (#183)
- servicediscovery timestamp localization issue (#217)
- ramuda bundling issue (#225)

13.56 [0.0.78] - 2017-03-06

13.56.1 Added

- moved hocon config_reader to plugin (#150)
- split gcdt_lookups plugin from config_reader (#150)
- improved slack plugin (webhooks, consolidated msgs) (#219)
- extracted bundle step into bundler plugin (#150)

13.57 [0.0.77] - 2017-02-20

13.57.1 Added

- moved to std python logging + activate DEBUG logs via `-v` (#175)
- std. gcdt lifecycle (#152)
- removed glomex_utils as installation dependency (#152)
- cloudformation utilities need awsclient (see kumo documentation) (#152)
- plugin mechanism (#152)
- moved datadog and slack reporting functionality to gcdt-plugins (#152)
- cmd dispatcher + testable main modules + tests (#152)
- migrated boto_session to awsclient (#152)

13.58 [0.0.76] - 2017-01-30

13.58.1 Added

- ramuda replaced git short hash in target filename with sha256 (#169)
- requirements.txt and settings_.conf optional for ramuda (#114)
- made boto_session a parameter in credstash (#177)

13.59 [0.0.75] - 2017-01-24

13.59.1 Added

- added gcdt installer (#201)
- gcdt outdated version warning (#155)
- moved docs from README to sphinx / readthedocs (PR194)
- pythonic dependency management (without pip-compile) (#178)
- removed glomex-utils dependency (#178)

13.59.2 Changed

- moved CHANGELOG.md to docs folder

13.60 [0.0.73] - 2017-01-09

13.60.1 Fixed

- (#194)

13.61 [0.0.64] - 2016-11-11

13.61.1 Fixed

- wrong boto client used when getting lambda arn

13.62 [0.0.63] - 2016-11-08

13.62.1 Fixed

- pre-hook fires before config is read (#165)

13.63 [0.0.62] - 2016-11-07

13.63.1 Added

- ramuda pre-bundle hooks

13.63.2 Fixed

- compress bundle.zip in ramuda bundle/deploy

13.64 [0.0.61] - 2016-11-07

13.64.1 Fixed

- moved build system to infra account (#160)

13.65 [0.0.60] - 2016-10-07

13.65.1 Added

- kumo now has the visualize cmd. Req. dot installation (#136).
- tenkai now has the slack notifications (#79).- FIX moved tests to pytest to improve cleanup after tests (#119).
- kumo now has parametrized hooks (#34).

13.65.2 Fixed

- moved tests to pytest to improve cleanup after tests (#119).
- ramuda rollback to previous version.
- kumo Parameter diffing does not work for aws coma-seperated inputs (#77).

- ramuda fail deployment on failing ping (#113).
- moved tests to pytest to improve cleanup after tests (#119).
- speedup tests by use of mocked service calls to AWS services (#151).

13.66 [0.0.57] - 2016-09-23

13.66.1 Added

- tenkai now supports execution of bash scripts before bundling, can be used to bundle packages at runtime.

13.66.2 Fixed

- tenkai now returns proper exit codes when deployment fails.

13.67 [0.0.55] - 2016-09-16

13.67.1 Added

- kumo utils EBS tagging functionality (intended for post hooks)
- kumo now supports host zones as a parameter for creating route53 records

13.68 [0.0.51] - 2016-09-05

13.68.1 Fixed

- kumo parameter diff now checks if stack has parameters beforehand

13.69 [0.0.45] - 2016-09-01

13.69.1 Added

- ramuda autowire functionality
- gcdt sends metrics and events to datadog

13.69.2 Fixed

- yugen will add invoke lambda permission for new paths in existing APIs

13.70 [0.0.35] - 2016-08-29

13.70.1 Added

- consolidated slack configuration to .gcdt files
- configuration for slack_channel

13.71 [0.0.34] - 2016-08-tbd

13.71.1 Fixed

- refactored yugen structure to yugen_main and yugen_core
- improved yugen testability and test coverage
- further improved ramuda test coverage

13.72 [0.0.33] - 2016-08-18

13.72.1 Added

- gcdt pull request builder

13.72.2 Fixed

- refactored tenkai structure to tenkai_main and tenkai_core
- improved tenkai testability and test coverage
- refactored ramuda structure to ramuda_main and ramuda_core
- improved ramuda testability and test coverage

13.73 [0.0.30] - 2016-08-02

13.73.1 Added

- refactored kumo structure to kumo_main and kumo_core
- improved kumo testability and test coverage
- Rate limiting when preview with empty changeset (#48)

13.73.2 Removed

- kumo validate
- kumo scaffold

13.74 [0.0.29] - 2016-07-21

13.74.1 Added

- bump glomex-utils to 0.0.11

13.74.2 Fixed

- create_stack was broken

13.75 [0.0.26] - 2016-07-19

13.75.1 Added

- kumo now supports stack policies, see README for details
- kumo now displays changes in CloudFormation template parameters

13.75.2 Fixed

- prettify output
- removed debug output
- removed some unnecessary import validations
- kumo will now exit when importing a cloudformation.py not from your current working directory

CHAPTER 14

Development of gcdt

14.1 Contributing

If you find any bugs or if you need new features please feel free to issue a pull request with your changes.

14.2 Issues and Feature Requests

Please open a GitHub issue for any bug reports and feature requests.

14.3 Common for all Tools

- All tools imply that your working directory is the directory that contains the artifact you want to work with.
- Furthermore you are responsible for supplying a valid set of AWS credentials. A good tool is aws-mfa
- You you need to set an environment variable "ENV" which indicates the account/staging area you want to work with. This parameter tells the tools which config file to use. Basically something like gcdt_$(ENV).json is evaluated in the configuration component.

14.4 Installing the development version locally

To install your local development version (after checkout):

```
$ pip install -e .
```

use pip to install the dev requirements:

```
$ pip install -r requirements_dev.txt
```

14.5 Running Unit-Tests

Use the pytest test-runner to run the gcdt unit tests. A few tests (with '_aws' in the file name) need AWS. Please turn on your VPN and set the `AWS_DEFAULT_PROFILE`, `ENV`, and `ACCOUNT` environment variables. Details here: https://confluence.glomex.com/display/OPSSHARED/Deployment+on+AWS.

You need to install the development version of this package so you can run the tests:

```
$ pip install -e .
```

```
$ export AWS_DEFAULT_PROFILE=superuser-dp-dev
$ export ENV=DEV
$ export ACCOUNT=dp # => or your team account
```

Note: You need to enter an MFA code to run the tests.

```
$ python -m pytest tests/test_kumo*
```

Please make sure that you do not lower the gcdt test coverage. You can use the following command to make sure:

```
$ python -m pytest --cov gcdt tests/test_ramuda*
```

This requires the `coverage` package (included in the `requirements_dev.txt` file):

```
$ pip install -r requirements_dev.txt
```

14.6 Mock calls to AWS services

For testing gcdt together with botocore and AWS services we use placebo_awsclient (a tool based on the boto maintainers placebo project). The way placebo_awsclient works is that it is attached to the botocore session and used to record and later playback the communication with AWS services.

The recorded json files for gcdt tests are stored in 'tests/resources/placebo_awsclient'.

gcdt testing using placebo playback is transparent (if you know how to run gcdt tests nothing changes for you).

To record a test using placebo **(first remove old recordings if any)**:

```
$ rm -rf tests/resources/placebo_awsclient/tests.test_tenkai_aws.test_tenkai_exit_
↪codes/
$ export PLACEBO_MODE=record
$ python -m pytest -vv --cov-report term-missing --cov gcdt tests/test_tenkai_aws.
↪py::test_tenkai_exit_codes
```

To switch off placebo record mode and use playback mode:

```
$ export PLACEBO_MODE=playback
```

To run the tests against AWS services (without recording) use `normal` mode:

```
$ export PLACEBO_MODE=normal
```

Please note:

- prerequisite for placebo to work is that all gcdt tools support that the awsclient is handed in as parameter (by the test or main). If a module creates its own botocore session it breaks gcdt testability.

- in order to avoid merging placebo json files please **never record all tests (it would take to long anyway)**. only record aws tests which are impacted by your change.
- gcdt testing using placebo works well together with aws-mfa.
- if you record the tests twice the json files probably get messed up. Please do not do this.
- Please commit the placebo files in a separate commit. This makes reviewing of pull requests easier.

14.7 documenting gcdt

For gcdt we need documentation and we publish it on Readthedocs. Consequently the tooling is already set like sphinx, latex, ... We would like to use markdown instead of restructured text so we choose recommonmark.

Detailed information on using markdown and sphinx

14.7.1 Installation of docu tools

```
$ pip install -r requirements_docs.txt
```

If you need to create the pdf docu install pdflatex (... you also need texlive!).

```
$ brew cask install mactex
```

14.7.2 build docu

In order to build the html and pdf version of the documentation

```
$ make html
$ make latexpdf
```

14.7.3 Release docu to Readthedocs

To release the documentation to Readthedocs most of the time there are no additional steps necessary. Just connect your rtfd account to your github repo.

14.7.4 Initialize api docu

We used the sphinx-apidoc tool to create the skeleton (80_gcdt_api.rst) for gcdt' api documentation.

```
$ sphinx-apidoc -F -o apidocs gcdt
```

14.8 Implementation details

This section is used to document gcdt implementation for gcdt developers and maintainers.

14.8.1 configuration

note: gcdt design has a section on openapi, too

configuration in gcdt is implemented in a few places:

- openapi functionality in gcdt (in 'gcdt/gcdt_openapi.py')
- openapi based validation for each tool (in 'gcdt_/openapi_.yaml' and 'gcdt_/plugin.py')
- the functionality to create docs from openapi specs

actual tool config	actual command	need to run	need to run
via config-reader	is non-config-command	incept_defaults	validate_config
yes	yes	yes	yes
yes	no	yes	yes
no	yes	yes *)	no
no	no	no	no

*) The above table shows that there is a case where we run a `non-config-command` and do not have configuration from the config reader. In this case we still need the defaults but the defaults alone might not be a valid configuration. To make this case work the `incept-defaults` functionality needs to disable the config validation for the impacted configuration part.

14.9 gcdt design

14.9.1 Design Goals

- support development teams with tools and templates
- ease, simplify, and master infrastructure-as-code

14.9.2 Design Principles

- write testable code
- tests need to run on all accounts (not just dp account)
- make sure additions and changes have powerful tests
- use pylint to increase your coding style
- we adhere to Semantic Versioning.

14.9.3 Design Decisions

In this section we document important design decisions we made over time while maintaining gcdt.

Use botocore over boto3

With botocore and boto3 AWS provides two different programmatic interfaces to automate interaction with AWS services.

One of the most noticeable differences between botocore and boto3 is that the client objects:

1. require parameters to be provided as `**kwargs`
2. require the arguments typically be provided as `CamelCased` values.

For example::

```
ddb = session.create_client('dynamodb')
ddb.describe_table(TableName='mytable')
```

In boto3, the equivalent code would be::

```
layer1.describe_table(table_name='mytable')
```

There are several reasons why this was changed in botocore.

The first reason was because we wanted to have the same casing for inputs as well as outputs. In both boto3 and botocore, the response for the `describe_table` calls is::

```
{'Table': {'CreationDateTime': 1393007077.387,
           'ItemCount': 0,
           'KeySchema': {'HashKeyElement': {'AttributeName': 'foo',
                                            'AttributeType': 'S'}},
           'ProvisionedThroughput': {'ReadCapacityUnits': 5,
                                     'WriteCapacityUnits': 5},
           'TableName': 'testtable',
           'TableStatus': 'ACTIVE'}}
```

Notice that the response is `CamelCased`. This makes it more difficult to round trip results. In many cases you want to get the result of a `describe*` call and use that value as input through a corresponding `update*` call. If the input arguments require `snake_casing` but the response data is `CamelCased` then you will need to manually convert all the response elements back to `snake_case` in order to properly round trip.

This makes the case for having consistent casing for both input and output. Why not use `snake_casing` for input as well as output?

We choose to use `CamelCasing` because this is the casing used by AWS services. As a result, we don't have to do any translation from `CamelCasing` to `snake_casing`. We can use the response values exactly as they are returned from AWS services.

This also means that if you are reading the AWS API documentation for services, the names and casing referenced there will match what you would provide to botocore. For example, here's the corresponding API documentation for `dynamodb.describe_table <http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_DescribeTable.html>__`.

14.9.4 Use pytest over nose

For many years py.test and nose coexisted as Python unit test frameworks in addition to std. Python unittest. Nose was developed by Mozilla and was popular for quite some time. In 2015 Mozilla switched from nose to pytest.

http://mathieu.agopian.info/presentations/2015_06_djangocon_europe/

There are many arguments in favour of pytest. For us the most important is pytest fixtures which provides us with a reliable and reusable mechanism to prepare and cleanup resources used during testing.

14.9.5 Capturing of log output during test

In our tests we use a `logcapture` fixture from gcdt_testtools.helpers to capture log output. The fixture use the textfixtures package under the hood.

use it like this:

```
logcapture.check(
    ('root', 'INFO', 'a message'),
    ('root', 'ERROR', 'another error'),
)
```

or:

```
records = list(logcapture.actual())
assert records[0][2] == 'a message'
```

details here: http://testfixtures.readthedocs.io/en/latest/logging.html

14.9.6 Use Sphinx, Readthedocs, and Markdown for documentation

Many, many documentation tools populate this space since it is so easy to come up with something. However for Open Source projects Readthedocs is the dominant platform to host the documentation.

The Sphinx is the Python std. docu tool. In combination with markdown tools set is a very convenient way to create Readthedocs conform documentation.

14.9.7 Keep a changelog

We where already keeping a changelog which "almost" followed the guide. In June 2017 we decided to make the format more explicit.

The gcdt changelog format is defined here: http://keepachangelog.com/en/1.0.0/

14.9.8 Use docopt to build the command line interface

There is a never-ending discussion going about pros and cons of CLI tools for Python. Some of these tools are contained in the Python std. library, some are independent open source library additions. At the moment the most popular tools are Optparse, Argparse, Click, and Docopt

https://www.youtube.com/watch?v=pXhcPJK5cMc

We decided to use docopt for out command line interface because it is simple and very flexible. In addition we developed a `dispatch mechanism` to ease the docopt usage and to make the gcdt CLI commands testable.

14.9.9 Using Maya: Datetimes for Humans

We had some issues in the past using datetimes correctly across different timezones and locales. glomex SRE team took an initiative to improve the situation and we. We looked into pytz and maya. Maya had a convincing offering and is maintained by Kenneth Reitz so we decided to use it for datetime handling within gcdt.

14.9.10 Plugin mechanism

gcdt uses entry points similar to pluggy to find installed plugins

For communication with the plugin we use Blinker signals. This helps us to decouple the gcdt code base from plugin code and vice-versa. Blinker is of cause only one way to do that. Blinker is fast, simple, well documented, etc. so there are some popular frameworks using it (Flask, Pelican, . . .).

14.9.11 Config handling using openapi

A wide area of gcdt functionality is related to configuration:

- default values for tools and plugins
- validation of configuration
- scaffolding of minimal (required properties) configuration
- sample of complete configuration
- documentation of configuration

To cover all this gcdt configuration usecases we decided to use the `openapi` specifications. Using openapi alows us to build on existing workflows and tooling.