Baikonur

Apr 05, 2021

Contents

1 What is Baikonur? **3**

2 Contents **5**

2.1 FAQ . . . 5
2.2 Baikonur Project repository index . . . 5
2.3 Knowledge Base . . . 7
2.4 Baikonur Logging with Amazon Kinesis and AWS Lambda . . . 7
2.5 eden: Amazon ECS Dynamic Environment Manager . . . 10

3 Indices and tables **21**

BAIKONU

CHAPTER 1

What is Baikonur?

Baikonur is an open source project that includes Terraform Modules, knowledge base, various infrastructure tools and more.

Baikonur started as an internal project at CyberAgent, Inc. in 2018, and quickly grew to over 50 modules. We started from open sourcing Baikonur started from releasing select most popular and unique internal Terraform Modules.

Terraform Modules released here are also published to Terraform Public Module Registry.

CHAPTER 2

Contents

2.1 FAQ

2.1.1 Why is this project named “Baikonur”?

Baikonur project name comes from Baikonur Cosmodrome, a spaceport located in Kazakhstan known for launching the first artificial satellite and the first human spaceflight. Hence, project logo is a space rocket.

The reason why we chose a space theme in the first place is quite simple - we followed Ansible Galaxy and GitHub Universe.

2.1.2 Is this project limited to Terraform Modules?

Absolutely not. Although Baikonur Project started as a platform for open sourcing internally used Terraform Modules, we have been creating Python libraries, CLI tools, and we are preparing to publish our infrastructure Knowledge Base.

2.1.3 Do I have to be a CyberAgent employee to become a contributor/maintainer/core member?

You do not *have to*. :)

2.2 Baikonur Project repository index

2.2.1 Baikonur Logging with Amazon Kinesis and AWS Lambda Terraform Modules

lambda-kinesis-to-fluent

Transfer data from Kinesis Data Streams to Fluent endpoint

lambda-kinesis-to-s3

Save data from Kinesis Data Streams to S3

lambda-kinesis-to-es

Add data from Kinesis Data Streams to Elasticsearch

lambda-es-cleaner

Automatically clean old indices in Elasticsearch Service

lambda-kinesis-forward

Kinesis Data Streams to Kinesis Data Streams forwarder/router module

2.2.2 eden: Amazon ECS Dynamic Environment Manager

aws-eden-cli

eden CLI

lambda-eden-api

eden API

2.2.3 Tools and libraries

amazon_kinesis_utils

A Python library of useful utilities for Amazon Kinesis. Extensively used by *logging modules*.

aws-eden-core

Internal library for *eden modules*.

2.2.4 Other Terraform Modules for AWS

iam-nofile

Module to make it easier to create AWS IAM Roles. You can write role policy document with inline(heredoc) syntax, so you do not have to use template rendering to use variables.

fargate-scheduled-task

Create ECS scheduled tasks with CloudWatch Events easily (for batch apps, or anything else that runs on routine)

2.3 Knowledge Base

Under construction.

2.4 Baikonur Logging with Amazon Kinesis and AWS Lambda

2.4.1 Prerequisites

Why Kinesis?

Kinesis Data Streams is a high-throughput, low latency, fully managed service for working with streaming data. A single shard provides 1 MB/s read and 2 MB/s write capacity. Data put to a Kinesis Data Stream is saved for a period specified in the data retention period parameter. Thus, for capacity planning, it is enough to know peak data throughput.

Note: Kinesis Data Streams do not have auto-scaling capabilities. You can have your Kinesis Data Stream shards scale automatically by using amazon-kinesis-scaling-utils.

Why Kinesis with Lambda?

Using Lambda functions to process data on Kinesis Streams (with event source mapping) reduces the amount of code and resources you have to manage:

1. With Lambda, you do not have to manage servers, OS, etc.
2. No need to write the logic for reading data from Kinesis Data Stream. Data batches are passed to Lambda functions on execution via `event` object.
3. Event source mapping manages processed data position on Kinesis Data Stream for you. It keeps track of until what position on each shard data is already successfully processed. The position is only updated if Lambda finishes without errors.
3. If a Lambda function mapped to a Kinesis Data Stream finishes with an error, it is executed again with the same batch until execution succeeds or data expires. Records are removed from Kinesis Data Stream only when the data retention period for those records expires.
    - Thus, you do not have to write retry logic in Lambda. Although, it may be a good idea to save data that is failing after n retries to a queue (or S3) and return without errors so that the data processing pipeline does not stop.

Why use Kinesis and Lambda for logging?

Kinesis and Lambda combination can replace self-managed log clusters with managed services while retaining control of log procession.

By using Kinesis with Lambda, we can create a modular, extendable, scalable logging architecture. Log transfer reliability may improve as well: data written to a Kinesis stream successfully will not get lost.

2.4.2 Common Schema requirements

"Baikonur Logging architecture" is any architecture using Kinesis Data Streams in conjunction with one or more of the following Baikonur Logging Lambda modules:

- terraform-aws-lambda-kinesis-forward
- terraform-aws-lambda-kinesis-to-es
- terraform-aws-lambda-kinesis-to-s3
- terraform-aws-lambda-kinesis-to-fluent

These modules have the following Common Schema requirements:

- All data must be JSON. The root element type must be an object.
- All data must include the following keys:
  - Data type identifier. (default key name: `log_type`)
  - Any unique identifier, e.g. `uuid.uuid4()` (default key name: `log_id`)
  - Any timestamp supported by dateutil. (default key name: `time`)

Note: All key names are customizable.

Common Schema requirements allow us for:

1. Easier parsing
2. Better interoperability between different Lambda modules.
   - You can attach different modules to the same Kinesis Data Stream.
3. Ability to create behavior based on keys that are part of Common Schema requirements:
   - One of the most important features is the ability to filter logs based on data type field value.
   - terraform-aws-lambda-kinesis-to-s3 requires log_id to ensure filename uniqueness and time key to separate logs by date.
   - terraform-aws-lambda-kinesis-to-es requires time key to create daily indices in Elasticsearch (e.g. index-20200314).

Note: As long as data meets Common Schema requirements, architectures and modules described in this documentation are applicable to any data transfer that needs to be fast and reliable, for example, inter-microservice communication.

2.4.3 Architecture examples

One stream - one destination

Create a Kinesis Data Stream for each destination. For example, if we want to save all log data to S3 and only some to Elasticsearch, create two streams, deploy terraform-aws-lambda-kinesis-to-s3 and terraform-aws-lambda-kinesis-to-es modules and map them to the respective streams:

```
   Kinesis API
       v
[App] ---> [KDS "s3"] ---> [kinesis-to-s3] ---> S3
      \
       ---> [KDS "es"] ---> [kinesis-to-es] ---> ES
```

If you want to save logs to both S3 and Elasticsearch, write data to both streams.

One stream - multiple destinations

In the example above, we have to write logs we want to save to Elasticsearch to both streams. We can further improve this by adding terraform-aws-lambda-kinesis-to-s3 to stream for Elasticsearch as well:

```
   Kinesis API
       v
[App] ---> [KDS "s3"] ---> [kinesis-to-s3] ---> S3
      \
       ---> [KDS "es"] ---> [kinesis-to-es] ---> ES
                      \
                       ---> [kinesis-to-s3] ---> S3
```

Now we write each log event at most once.

Kinesis routing pattern

Write data to a single Kinesis stream (a “router”). Create multiple output streams, each for a destination. We can use forwarder modules (terraform-aws-lambda-kinesis-forward) with whitelists to create an architecture similar to the Publish-subscribe pattern, where a topic is a value in the type field, and each output stream represents a subscription group:

```
   Kinesis API
       v
[App] ---> [KDS "router"] ---> [kinesis-forward] ---> [KDS "A"]
                          \
                           ---> [kinesis-forward] ---> [KDS "B"]
                            \
                             --> [kinesis-forward] ---> [KDS "C"]
```

This pattern may also be useful for inter-microservice communication.

Each of output streams may have their own Lambda modules or subscribers:

```
   Kinesis API
       v
[App] ---> [KDS "router"] ---> [kinesis-forward] ---> [KDS "A"] ---> [S3]
                          \
                           ---> [kinesis-forward] ---> [KDS "B"] ---> [ES]
                            \
                             --> [kinesis-forward] ---> [KDS "C"] <--- [External␣
↪subscriber]
```

Kinesis routing pattern with CloudWatch Logs subscription filters

In addition to *Kinesis Routing pattern*, use CloudWatch Logs subscription filters to write data to the "router" stream. Doing so will free you from having to write PutRecord/PutRecords logic in your application if you already output logs to CloudWatch. For instance, if you are using awslogs logging driver in ECS, using subscription filter may look like this:

```
 stdout->awslogs      Subscription filter
       v                    v
[App] ---> [CloudWatch Logs] ---> [KDS "router"] ---> [kinesis-forward] ---> [KDS "A
↪"] ---> [S3]
                                              \
                                               ---> [kinesis-forward] ---> [KDS "B
↪"] ---> [ES]
                                                \
                                                 --> [kinesis-forward] ---> [KDS "C
↪"] <--- [External subscriber]
```

2.5 eden: Amazon ECS Dynamic Environment Manager

Clone Amazon ECS environments easily. Provide eden with a Amazon ECS service and eden will clone it.

eden is fast: create/delete commands should not take more than 5 seconds to run.

2.5.1 Why?

eden is made for use cases, where many "preview" development environments are needed, but databases and other resources may be shared.

2.5.2 How?

eden clones Task Definition, ECS Service and Target Group from a reference service.

eden uses one common ALB for all cloned services to save money and time (it can take up to 5 minutes to create an ALB). eden creates a Target Group, clones a service, attaches created Target Group to common ALB, and creates a Route 53 A ALIAS record pointing at common ALB.

Resource creation order

1. ECS Task Definition
    - Cloned from reference service
2. ALB Target Group
    - Settings are cloned from the Target Group attached to reference service
3. ECS Service
    - Created in the same cluster as reference service
4. ALB Listener Rule
    - Host Header rule

5. Route 53 A ALIAS record
   - Points at common ALB
6. An entry is added to environments JSON file

Note: Resource deletion is performed in reverse order. Both creation and deletion should take no more than 5 seconds.

2.5.3 Prerequisites

1. Environments JSON file in a S3 bucket
   - Structure and details are described *here*
2. A reference ECS Service with Target Group Attached
3. A common ALB for services managed by eden
   - Will be reused by all environments with Host Header Listener Rules
   - Separate from what reference service uses
   - Must have HTTPS Listener
   - Listener must have wildcard certificate for target dynamic zone
4. Simple ALB usage
   - No multiple path rules etc.
   - One ALB per one ECS Service

2.5.4 Preparations

Under construction

2.5.5 Environments JSON file

The Environments JSON file is used to:

1. Check what environments exist and where their endpoints are
2. Tell client apps what is available

Environments JSON file example:

```
{
    "environments": [
        {
            "env": "dev",
            "name": "dev-dynamic-test",
            "api_endpoint": "api-test.dev.example.com"
        }
    ]
}
```

Example above presumes `config_update_key = api_endpoint`.

You can create multiple environments with the same names, but with profiles with different `config_update_key` settings to have multiple endpoints within a single environment. For example, you may want to have an API, administration tool, and a frontend service created as a single environment.

Let's say we have three profiles, `api`, `admin`, and `frontend`. These profiles are pre-configured with `config_update_key` equal to `api_endpoint`, `admin_endpoint`, `frontend_endpoint` respectively.

```
$ eden create -p api --name foo --image-uri xxxxxxxxxx.dkr.ecr.ap-northeast-1.
↪amazonaws.com/api:latest
$ eden create -p admin --name foo --image-uri xxxxxxxxxx.dkr.ecr.ap-northeast-1.
↪amazonaws.com/admin:latest
$ eden create -p frontend --name foo --image-uri xxxxxxxxxx.dkr.ecr.ap-northeast-1.
↪amazonaws.com/frontend:latest
```

Your environment file could look like this:

```
{
    "environments": [
        {
            "env": "dev",
            "name": "dev-dynamic-test",
            "api_endpoint": "api-test.dev.example.com",
            "admin_endpoint": "admin-test.dev.example.com",
            "frontend_endpoint": "test.dev.example.com"
        }
    ]
}
```

Multiple endpoints exist in a single JSON object within a same environment name in environments JSON file. This environment entry will be removed from environments JSON file when last endpoint is deleted.

Warning: When working with multiple endpoints in a single environment like in example above, keep in mind that endpoint creation/deletion time difference may result in incomplete environments (not having all necessary endpoints).

2.5.6 CLI and API

eden is provided in two flavors: CLI and API.

We recommend trying eden out with CLI, and when you feel you are ready to make eden part of your CI/CD pipeline, switch to API. Please note that you will need to use CLI to push *profiles* for API.

Installing eden CLI

```
$ pip3 install aws-eden-cli

$ eden -h
usage: eden [-h] {create,delete,ls,config} ...

ECS Dynamic Environment Manager. Clone Amazon ECS environments easily.

positional arguments:
```

(continues on next page)

(continued from previous page)

```
  {create,delete,ls,config}
    create              Create environment or deploy to existent
    delete              Delete environment
    ls                  List existing environments
    config              Configure eden

optional arguments:
  -h, --help            show this help message and exit
```

Hint: you can use -h on subcommands as well:

```
$ eden config -h
 usage: eden config [-h] {setup,check,push,remote-rm} ...

positional arguments:
  {setup,check,push,remote-rm}
    setup               Setup profiles for other commands
    check               Check configuration file integrity
    push                Push local profile to DynamoDB for use by eden API
    remote-rm           Delete remote profile from DynamoDB

optional arguments:
  -h, --help            show this help message and exit

$ eden config push -h
usage: eden config push [-h] [-p PROFILE] [-c CONFIG_PATH] [-v]
                        [--remote-table-name REMOTE_TABLE_NAME]

optional arguments:
  -h, --help            show this help message and exit
  -p PROFILE, --profile PROFILE
                        profile name in eden configuration file
  -c CONFIG_PATH, --config-path CONFIG_PATH
                        eden configuration file path
  -v, --verbose
  --remote-table-name REMOTE_TABLE_NAME
                        profile name in eden configuration file
```

Configuring eden CLI

Let’s create a profile to work with, so we won’t have to specify all the parameters every time:

```
$ eden config setup --endpoint-s3-bucket-name servicename-config
$ eden config setup --endpoint-s3-key endpoints.json
$ eden config setup --endpoint-name-prefix servicename-dev
$ eden config setup --endpoint-update-key api_endpoint
$ eden config setup --endpoint-env-type dev
$ eden config setup --domain-name-prefix api
$ eden config setup --dynamic-zone-id Zxxxxxxxxxxx
$ eden config setup --master-alb-arn arn:aws:elasticloadbalancing:ap-northeast-
↪1:xxxxxxxxxxxx:loadbalancer/app/dev-alb-api-dynamic/xxxxxxxxxx
$ eden config setup --name-prefix dev-dynamic
$ eden config setup --reference-service-arn arn:aws:ecs:ap-northeast-
↪1:xxxxxxxxxxxx:service/dev/dev01-api
$ eden config setup --target-cluster dev
```

Configuration is saved to `~/.eden/config`. Commands above created a "default" profile:

```
$ cat ~/.eden/config
[api]
name_prefix = dev-dynamic
reference_service_arn = arn:aws:ecs:ap-northeast-1:xxxxxxxxxxxx:service/dev/dev01-api
target_cluster = dev
domain_name_prefix = api
master_alb_arn = arn:aws:elasticloadbalancing:ap-northeast-
↪1:xxxxxxxxxxxx:loadbalancer/app/dev-alb-api-dynamic/xxxxxxxxxx
dynamic_zone_name = dev.example.com.
dynamic_zone_id = Zxxxxxxxxxxxx
config_bucket_name = servicename-config
config_bucket_key = endpoints.json
config_update_key = api_endpoint
config_env_type = dev
config_name_prefix = servicename-dev
target_container_name = api
```

Don't forget to check configuration file integrity:

```
$ eden config check
No errors found
```

Profiles in eden

You can create multiple profiles in configuration and specify a profile to use with `-p profile_name` for all commands.

```
$ eden config check -p api
No errors found
```

We can push profiles to DynamoDB for use by eden API:

```
$ eden config push -p api
Waiting for table creation...
Successfully pushed profile api to DynamoDB
```

Note: If eden table does not exist, eden CLI will create it

Use the same command to overwrite existing profiles (push to existing profile will result in overwrite):

```
$ eden config push -p api
Successfully pushed profile api to DynamoDB table eden
```

Use remote-rm to delete remote profiles:

```
$ eden config remote-rm -p api
Successfully removed profile api from DynamoDB table eden
```

Execute commands

Create an environment:

```
$ eden create -p api --name foo --image-uri xxxxxxxxxx.dkr.ecr.ap-northeast-1.
↪amazonaws.com/api:latest
Checking if image xxxxxxxxxx.dkr.ecr.ap-northeast-1.amazonaws.com/api:latest exists
Image exists
Retrieved reference service arn:aws:ecs:ap-northeast-1:xxxxxxxxxx:service/dev/api
Retrieved reference task definition from arn:aws:ecs:ap-northeast-1:xxxxxxxxxx:task-
↪definition/api:20
Registered new task definition: arn:aws:ecs:ap-northeast-1:xxxxxxxxxx:task-definition/
↪dev-dynamic-api-foo:1
Registered new task definition: arn:aws:ecs:ap-northeast-1:xxxxxxxxxx:task-definition/
↪dev-dynamic-api-foo:1
Retrieved reference target group: arn:aws:elasticloadbalancing:ap-northeast-
↪1:xxxxxxxxxx:targetgroup/api/xxxxxxxxxxxx
Existing target group dev-dynamic-api-foo not found, will create new
Created target group arn:aws:elasticloadbalancing:ap-northeast-
↪1:xxxxxxxxxx:targetgroup/dev-dynamic-api-foo/xxxxxxxxxxxx
ELBv2 listener rule for target group arn:aws:elasticloadbalancing:ap-northeast-
↪1:xxxxxxxxxx:targetgroup/dev-dynamic-api-foo/xxxxxxxxxxxx and host api-foo.dev.
↪example.com does not exist, will create new listener rule
ECS Service dev-dynamic-api-foo does not exist, will create new service
Checking if record api-foo.dev.example.com. exists in zone Zxxxxxxxxx
Successfully created CNAME: api-foo.dev.example.com -> dev-alb-api-dynamic-297517510.
↪ap-northeast-1.elb.amazonaws.com
Updating config file s3://example-com-config/endpoints.json, environment example-api-
↪foo: nodeDomain -> api-foo.dev.example.com
Existing environment not found, adding new
Successfully updated config file
Successfully finished creating environment dev-dynamic-api-foo
```

Note: Create and delete commands update remote state DynamoDB Table. As with `eden config push`, table will be created for you if it does not exist.

Check creation:

```
$ eden ls
Profile api:
dev-dynamic-api-foo api-foo.dev.example.com (last updated: 2019-11-20T19:44:10.179760)
```

Note: This list is generated from remote state DynamoDB Table and not *environments JSON* file. Last updated timestamp is updated on creation and deploys as well.

Delete environment and check deletion:

```
$ eden delete -p api --name foo
Updating config file s3://example-com-config/endpoints.json, delete environment␣
↪example-api-foo: nodeDomain -> api-foo.dev.example.com
Existing environment found, and the only optional key is nodeDomain,deleting␣
↪environment
Successfully updated config file
Checking if record api-foo.dev.example.com. exists in zone Zxxxxxxxxx
Found existing record api-foo.dev.example.com. in zone Zxxxxxxxxx
Successfully removed CNAME record api-foo.dev.example.com
ECS Service dev-dynamic-api-foo exists, will delete
```

(continues on next page)

(continued from previous page)

```
Successfully deleted service dev-dynamic-api-foo from cluster dev
ELBv2 listener rule for target group arn:aws:elasticloadbalancing:ap-northeast-
↪1:xxxxxxxxxx:targetgroup/dev-dynamic-api-foo/xxxxxxxxxxxx and host api-foo.dev.
↪example.com found, will delete
Deleted target group arn:aws:elasticloadbalancing:ap-northeast-
↪1:xxxxxxxxxx:targetgroup/dev-dynamic-api-foo/xxxxxxxxxxxx
Deleted all task definitions for family: dev-dynamic-api-foo, 1 tasks deleted total
Successfully finished deleting environment dev-dynamic-api-foo

$ eden ls
No environments available
```

2.5.7 Moving to API

Both CLI and API manage their state in a DynamoDB Table. This table is only created by CLI. Furthermore, API can only use “remote profiles”, saved in state table. Before running API, make sure you pushed a profile to use with API by running `eden config --push`. If this table does not exist during API creation in Terraform, terraform apply will fail.

API internals

eden API consists of:

1. Lambda function (the API itself)
2. API Gateway with API key for protecting API
3. DynamoDB Table for state management
   - Default table name is eden.

Creating eden API with Terraform

```
module "eden_api" {
  source  = "baikonur-oss/lambda-eden-api/aws"
  version = "0.2.0"

  lambda_package_url = "https://github.com/baikonur-oss/terraform-aws-lambda-eden-api/
↪releases/download/v0.2.0/lambda_package.zip"
  name               = "eden"

  # eden API Gateway variables
  api_acm_certificate_arn = "${data.aws_acm_certificate.wildcard.arn}"
  api_domain_name         = "${var.env}-eden.${data.aws_route53_zone.main.name}"
  api_zone_id             = "${data.aws_route53_zone.main.zone_id}"

  endpoints_bucket_name = "somebucket"

  dynamic_zone_id = "${data.aws_route53_zone.dynamic.zone_id}"
}
```

Warning: DynamoDB table for state management is created by eden CLI. Make sure to run `eden config --push` with success at least once before terraform apply.

With multiple profiles, one eden API instance is enough for one account/region. Refer to *profile section* for more details.

eden API commands

eden has only two API commands: create and delete.

GET /api/v1/create

Required query parameters:

- name: environment name
- image_uri: ECR image URI to deploy, must be already pushed and must be in the same account (eden API will check for image availability before deploying)

Optional query parameters:

- profile: eden profile to use (default value = `default`). Profiles include all settings necessary. Profiles can be created with `eden config --push` command (*see here for details*).

GET /api/v1/delete

Required query parameters:

- name: environment name

Optional query parameters:

- profile: eden profile to use (default value = `default`). Profiles include all settings necessary. Profiles can be created with `eden config --push` command (*see here for details*).

eden API Keys

eden API Terraform module creates one API Key for you. You can check it from API Gateway console.

You will need to specify this key to access API.

Key must be provided as an HTTP header:

```
x-api-key: YOURAPIKEY
```

API example

Let's run create API (with a remote profile called `api`):

```
curl https://eden.example.com/api/v1/create?name=test-create&image_uri=xxxxxxxxxxxx.
↪dkr.ecr.ap-northeast-1.amazonaws.com/servicename-api-dev:latest&profile=api -H "x-
↪api-key:YOURAPIKEY"
```

Now let's look at logs that API Lambda Function has produced:

```
2019-04-08T20:32:05.151Z INFO     [main.py:check_cirn:382] Checking if image␣
↪xxxxxxxxxxxx.dkr.ecr.ap-northeast-1.amazonaws.com/servicename-api-dev:latest exists
2019-04-08T20:32:05.270Z INFO     [main.py:check_cirn:401] Image exists
2019-04-08T20:32:05.446Z INFO     [main.py:create_env:509] Retrieved reference␣
↪service arn:aws:ecs:ap-northeast-1:xxxxxxxxxxxx:service/dev/dev01-api
2019-04-08T20:32:05.484Z INFO     [main.py:create_task_definition:58] Retrieved␣
↪reference task definition from arn:aws:ecs:ap-northeast-1:xxxxxxxxxxxx:task-
↪definition/dev01-api:15
2019-04-08T20:32:05.557Z INFO     [main.py:create_task_definition:96] Registered new␣
↪task definition: arn:aws:ecs:ap-northeast-1:xxxxxxxxxxxx:task-definition/dev-
↪dynamic-test-create:1
2019-04-08T20:32:05.584Z INFO     [main.py:create_target_group:112] Retrieved␣
↪reference target group: arn:aws:elasticloadbalancing:ap-northeast-
↪1:xxxxxxxxxxxx:targetgroup/dev01-api/9c68a5f91f34d9a4
2019-04-08T20:32:05.611Z INFO     [main.py:create_target_group:125] Existing target␣
↪group dev-dynamic-test-create not found, will create new
2019-04-08T20:32:06.247Z INFO     [main.py:create_target_group:144] Created target␣
↪group
2019-04-08T20:32:06.310Z INFO     [main.py:create_alb_host_listener_rule:355] ELBv2␣
↪listener rule for target group arn:aws:elasticloadbalancing:ap-northeast-
↪1:xxxxxxxxxxxx:targetgroup/dev-dynamic-test-create/b6918e6e5f10389d and host api-
↪test.dev.example.com does not exist, will create new listener rule
2019-04-08T20:32:06.361Z INFO     [main.py:create_env:554] ECS Service dev-dynamic-
↪test-create does not exist, will create new service
2019-04-08T20:32:07.672Z INFO     [main.py:check_record:414] Checking if record api-
↪test.dev.example.com. exists in zone Zxxxxxxxxxxxx
2019-04-08T20:32:08.133Z INFO     [main.py:create_cname_record:477] Successfully␣
↪created ALIAS: api-test.dev.example.com -> dev-alb-api-dynamic-xxxxxxxxx.ap-
↪northeast-1.elb.amazonaws.com
2019-04-08T20:32:08.134Z INFO     [main.py:create_env:573] Successfully finished␣
↪creating environment dev-dynamic-test-create
```

As state is managed in a remote DynamoDB table, you can check creation using eden CLI:

```
$ eden ls
Profile api:
dev-dynamic-test-create api-test.dev.example.com (last updated: 2019-04-08T20:32:08.
↪134469)
```

Now let's delete this environment by running:

```
curl https://eden.example.com/api/v1/delete?name=test&profile=api -H "x-api-
↪key:YOURAPIKEY"
```

API Lambda logs will look like this:

```
2019-04-10T23:11:38.515Z INFO     [main.py:check_record:495] Checking if record api-
↪test.dev.example.com. exists in zone Zxxxxxxxxxxxx
2019-04-10T23:11:38.752Z INFO     [main.py:check_record:506] Found existing record␣
↪api-test.dev.example.com. in zone Zxxxxxxxxxxxx
2019-04-10T23:11:38.996Z INFO     [main.py:delete_cname_record:596] Successfully␣
↪removed ALIAS record api-test.dev.example.com
2019-04-10T23:11:39.245Z INFO     [main.py:delete_env:665] ECS Service dev-dynamic-
↪test exists, will delete
2019-04-10T23:11:39.401Z INFO     [main.py:delete_env:670] Successfully deleted␣
↪service dev-dynamic-test from cluster dev
```

(continues on next page)

(continued from previous page)

```
2019-04-10T23:11:39.573Z INFO     [main.py:delete_alb_host_listener_rule:397] ELBv2␣
↪listener rule for target group arn:aws:elasticloadbalancing:ap-northeast-
↪1:xxxxxxxxxxxx:targetgroup/dev-dynamic-test/xxxxxxxx and host api-test.dev.example.
↪com found, will delete
2019-04-10T23:11:40.483Z INFO     [main.py:delete_env:697] Deleted all task␣
↪definitions for family: dev-dynamic-test, 5 tasks deleted total
2019-04-10T23:11:40.483Z INFO     [main.py:delete_env:700] Successfully finished␣
↪deleting environment dev-dynamic-test
```

CHAPTER 3

Indices and tables

- genindex
- search