Stream processing with R in AWS

AWR, AWR.KMS, AWR.Kinesis (R packages) used in ECS

Gergely Daroczi

@daroczig

July 05, 2017
About me

3,633 contributions in the year before last

<table>
<thead>
<tr>
<th></th>
<th>Mar</th>
<th>Apr</th>
<th>May</th>
<th>Jun</th>
<th>Jul</th>
<th>Aug</th>
<th>Sep</th>
<th>Oct</th>
<th>Nov</th>
<th>Dec</th>
<th>Jan</th>
<th>Feb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fri</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Learn how we count contributions.

Less  |  |  |  |  |  |  |  |  |  |  |  |  |
More  |  |  |  |  |  |  |  |  |  |  |  |  |

3,470 contributions in the last year

<table>
<thead>
<tr>
<th></th>
<th>Mar</th>
<th>Apr</th>
<th>May</th>
<th>Jun</th>
<th>Jul</th>
<th>Aug</th>
<th>Sep</th>
<th>Oct</th>
<th>Nov</th>
<th>Dec</th>
<th>Jan</th>
<th>Feb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fri</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Learn how we count contributions.

Less  |  |  |  |  |  |  |  |  |  |  |  |  |
More  |  |  |  |  |  |  |  |  |  |  |  |  |

Lead R Developer

Director of Analytics
About me

Gergely Daroczi (@daroczig)

Stream processing using AWR

gitlab.com/cardcorp/AWR
About me

Gergely Daroczi (@daroczig)

Stream processing using AWR

gitlab.com/cardcorp/AWR
About me

Gergely Daroczi (@daroczig)

Stream processing using AWR
Stream Processing . . . Why R?

Gergely Daroczi (@daroczig)

Stream processing using AWR
gitlab.com/cardcorp/AWR
Stream Processing . . . Why AWS?

**Infrastructure**

The goal of Rapporter was to provide a frontline to R in **all modern browsers** running on **various platforms** [A] – let it be a desktop, a notebook, tablet, or a mobile phone. Users can access their data, reports and statistical tools stored in the Rapporter cloud from any place over the Internet and even co-operate collaboratively with other users and contributors.

A minor but useful part of the infrastructure is hosted by Zendesk that provides an extensible **knowledge base** and **support forum** [B].

All the requests and data packets sent by the clients to Rapporter servers hit our **Content Delivery Network** provider [C] first that would return all static content of the webpage cached at several locations around the world for improved response times. The CDN also operates as a front-line **firewall** and filters out some unwanted and potentially dangerous packets and queries [D] beside minimizing the risk of (Distributed) Denial-of-Service attacks. Users can optionally use Rapporter over a **secure channel** by HTTPS protocol [E], as the data transmitted to and from CloudFlare is encrypted on demand for improved security.

The dynamic content is mainly served by our **Ruby on Rails** [F] workers in the means of a cluster of thin servers [G] running inside of our private internal network. This **content management system** is made of several separate threads replying to user request via a **load balancing** reverse proxy [H] that also serves static content, plus JavaScript, CSS and image assets.

Although we try to do our best with deploying working code on production servers, we also collect possible Rails **error messages** with **Erubi** [I].

Another major part of our setup is the **HAProxy** cluster [J] of **Apache** based **Rworkers** [J], running within an enforced **AppArmor** [K] profile and optional **AppArmor** hats based on user privileges. This latter Linux kernel **security module** ensures that the users could not directly touch the disks or make connections to our databases – even if some malicious code would somehow escape our in-house developed **sandbox** called **sandboxR**. The dynamic hat option allows fine-grained control over the **hardware resources** on a per-user basis in the means of e.g. CPU power and memory limit, or network access. Please see some further **security considerations** below.

As we are using **R** for creating **complex** or one-time and temporary **reports** via a **Graphical User Interface** or the recently introduced **Application Programming Interface** called **Raplications**, our home-made internal **R** functions do not deal with any statistical problems, but rather provides an environment for the users to easily implement those. Rapporter is basically made of our open-source **rapport** and **pander** packages (please see below) beside the above described Rails front-end and hardened security tools, and the data, methods and results are bundled all in various **JSON** driven databases [M].

All our servers are running **Ubuntu LTS** [N] on 64 bit with a decent amount of memory and CPU cores optionally dedicated to VIP customers, and continuously **monitored** 24 hours a day, 7 days a week via the public **Pingdom** availability monitor [O] and a more detailed and technical, enterprise-class monitoring solution with thousands of metrics called **Zabbix** [P] – beside Google Analytics of course.

**Data storage** [M]

Although administering and maintaining several similar database engines might not make much sense in most setup, we use two **NoSQL** databases for **improved performance**. **CouchDB** is awesome for its disk-based **B-tree** views, simple attachment concept and eventual consistency scheme, while **MongoDB** makes the Rails models a lot more convenient to work with. **GlusterFS** is a network filesystem that stores R-generated images on a **replicated** and optionally distributed storage attached to the highly available Rails servers.

Gergely Daroczi (@daroczig)
Intro to Amazon Kinesis

Source: Kinesis Product Details
Intro to Amazon Kinesis Streams

Source: Kinesis Developer Guide
Intro to Amazon Kinesis Shards

Source: AWS re:Invent 2013
A Very Deep Learning

Capture & submit streaming data to Firehose

Firehose loads streaming data continuously into S3 and Redshift

Analyze streaming data using your favorite BI tools
A Very Deep Learning

Capture & submit streaming data to Firehose

Firehose loads streaming data continuously into S3 and Redshift

Analyze streaming data using your favorite BI tools
S4: Multiple Dispatch

Diagram showing the integration of AWS Lambda, Amazon EC2, Streams, Firehose, Amazon S3, and Amazon Redshift.
How to Communicate with Kinesis

Writing data to the stream:

- Amazon Kinesis Streams API, SDK
- Amazon Kinesis Producer Library (KPL) from Java
- flume-kinesis
- Amazon Kinesis Agent

Reading data from the stream:

- Amazon Kinesis Streams API, SDK
- Amazon Kinesis Client Library (KCL) from Java, Node.js, .NET, Python, Ruby

Managing streams:

- Amazon Kinesis Streams API (!)
Now We Need an R Client!

```r
> library(rJava)
> .jinit(classpath = list.files('~/Projects/AWR/inst/java/', full.names = TRUE))

> kc <- .jnew('com.amazonaws.services.kinesis.AmazonKinesisClient')
> kc$setEndpoint('kinesis.us-west-2.amazonaws.com', 'kinesis', 'us-west-2')

> sir <- .jnew('com.amazonaws.services.kinesis.model.GetShardIteratorRequest')
> sir$setStreamName('test_kinesis')
> sir$setShardId(.jnew('java/lang/String', '0'))
> sir$setShardIteratorType('TRIM_HORIZON')
> iterator <- kc$getShardIterator(sir)$getShardIterator()

> grr <- .jnew('com.amazonaws.services.kinesis.model.GetRecordsRequest')
> grr$setShardIterator(iterator)
> kc$getRecords(grr)$getRecords()
[1] "Java-Object{[SequenceNumber: 4956289416044944321533463710843135723243616650,
ApproximateArrivalTimestamp: Tue Jun 14 09:40:19 CEST 2016,
Data: java.nio.HeapByteBuffer[pos=0 lim=6 cap=6],PartitionKey: 42]}"

> sapply(kc$getRecords(grr)$getRecords(),
+ function(x)
+   rawToChar(x$getData()$array()))
[1] "foobar"
```
Managing Shards via the Java SDK

Let’s merge two shards:

```scala
> ms <- .jnew('com.amazonaws.services.kinesis.model.MergeShardsRequest')
> ms$setShardToMerge('shardId-000000000000')
> ms$setAdjacentShardToMerge('shardId-000000000001')
> ms$setStreamName('test_kinesis')
> kc$mergeShards(ms)
```

What do we have now?

```scala
> kc$describeStream(StreamName = 'test_kinesis')$getStreamDescription()$getShards()
[1] "Java-Object{[
{ShardId: shardId-000000000000, HashKeyRange: {StartingHashKey: 0, EndingHashKey: 170141183460469231731687303715884105727},
SequenceNumberRange: {
StartingSequenceNumber: 49562894160427143586954815717376297430913467927668719618,
EndingSequenceNumber: 4956289416044944321533463405178331491861162891747005005,
},
{ShardId: shardId-000000000001, HashKeyRange: {StartingHashKey: 170141183460469231731687303715884105728,
SequenceNumberRange: {
StartingSequenceNumber: 49562894160460594704752611652087392082504911751749828626,
EndingSequenceNumber: 49562894160460594704752611652087392082504911751749828626},
{ShardId: shardId-000000000002,
ParentShardId: shardId-000000000000,
AdjacentParentShardId: shardId-000000000001,
HashKeyRange: {StartingHashKey: 0, EndingHashKey: 3402823669209384634633746074317682},
SequenceNumberRange: {StartingSequenceNumber: 49562890499149767309704924344727019520,
EndingSequenceNumber: 49562890499149767309704924344727019520},
```

Gergely Daroczi (@daroczig)
Stream processing using AWR
https://gitlab.com/cardcorp/AWR
Amazon Kinesis Client Library

- An *easy-to-use* programming model for processing data

```java
java -cp amazon-kinesis-client-1.7.3.jar \
com.amazonaws.services.kinesis.multilang.MultiLangDaemon \
app.properties
```

- *Scalable* and *fault-tolerant* processing (checkpointing via DynamoDB)
- Logging and metrics in CloudWatch
- The **MultiLangDaemon** spawns processes written in any language, communication happens via JSON messages sent over stdin/stdout
- Only a few events/methods to care about in the consumer application:
  1. initialize
  2. processRecords
  3. checkpoint
  4. shutdown
Messages from the KCL

1. **initialize:**
   - Perform initialization steps
   - Write “status” message to indicate you are done
   - Begin reading line from STDIN to receive next action

2. **processRecords:**
   - Perform processing tasks (you may write a checkpoint message at any time)
   - Write “status” message to STDOUT to indicate you are done.
   - Begin reading line from STDIN to receive next action

3. **shutdown:**
   - Perform shutdown tasks (you may write a checkpoint message at any time)
   - Write “status” message to STDOUT to indicate you are done.
   - Begin reading line from STDIN to receive next action

4. **checkpoint:**
   - Decide whether to checkpoint again based on whether there is an error or not.
#!/usr/bin/r -i

while (TRUE) {

    ## read and parse JSON messages
    line <- fromJSON(readLines(n = 1))

    ## nothing to do unless we receive records to process
    if (line$action == 'processRecords') {

        ## process each record
        lapply(line$records, function(r) {

            business_logic(fromJSON(rawToChar(base64_dec(r$data))))
            cat(toJSON(list(action = 'checkpoint', checkpoint = r$sequenceNumber)))

        })
    }

    ## return response in JSON
    cat(toJSON(list(action = 'status', responseFor = line$action)))
}

Gergely Daroczi (@daroczig)  Stream processing using AWR  gitlab.com/cardcorp/AWR
#!/usr/bin/r -i

while (TRUE) {

## read and parse JSON messages
line <- fromJSON(readLines(n = 1))

## nothing to do unless we receive records to process
if (line$action == 'processRecords') {

## process each record
lapply(line$records, function(r) {

  business_logic(fromJSON(rawToChar(base64_dec(r$data))))
  cat(toJSON(list(action = 'checkpoint', checkpoint = r$sequenceNumber)))

})

}

## return response in JSON
cat(toJSON(list(action = 'status', responseFor = line$action)))
}
Get rid of the bugs and the boilerplate

> `install.packages('AWR.Kinesis')`
also installing the dependency 'AWR'

trying URL 'https://cloud.r-project.org/src/contrib/AWR_1.11.89.tar.gz'
Content type 'application/x-gzip' length 3125 bytes

trying URL 'https://cloud.r-project.org/src/contrib/AWR.Kinesis_1.7.3.tar.gz'
Content type 'application/x-gzip' length 3091459 bytes (2.9 MB)

* installing *source* package 'AWR' ...
** testing if installed package can be loaded
trying URL 'https://gitlab.com/cardcorp/AWR/repository/archive.zip?ref=1.11.89'
downloaded 58.9 MB
* DONE (AWR)

* installing *source* package 'AWR.Kinesis' ...
* DONE (AWR.Kinesis)
Add content to the boilerplate

Business logic coded in R (demo_app.R):

```r
library(AWR.Kinesis)
kinesis_consumer(processRecords = function(records) {
    flog.info(jsonlite:::toJSON(records))
})
```

Note: This is not something you should run in RStudio.
Add content to the boilerplate

Business logic coded in R (demo_app.R):

```r
library(AWR.Kinesis)
kinesis_consumer(processRecords = function(records) {
    flog.info(jsonlite::toJSON(records))
})
```

Note
---
This is not something you should run in RStudio.
Add content to the boilerplate

Business logic coded in R (demo_app.R):

```r
library(AWR.Kinesis)
kinesis_consumer(processRecords = function(records) {
  flog.info(jsonlite::toJSON(records))
})
```

Config file for the MultiLangDaemon (demo_app.properties):

```properties
executableName = ./demo_app.R
streamName = demo_stream
applicationName = demo_app
```

Start the MultiLangDaemon:

```
/usr/bin/java -cp AWR/java/*:AWR.Kinesis/java/*:.:/
   com.amazonaws.services.kinesis.multilang.MultiLangDaemon \
   ./demo_app.properties
```
library(futile.logger)
library(AWR.Kinesis)

kinesis_consumer(

    initialize = function()
    flog.info('Hello'),

    processRecords = function(records)
    flog.info(paste('Received', nrow(records), 'records from Kinesis')),

    shutdown = function()
    flog.info('Bye'),

    updater = list(
    list(1, function()
        flog.info('Updating some data every minute')),
    list(1/60*10, function()
        flog.info(paste(
            'This is a high frequency updater call',
            'running every 10 seconds'))),

    checkpointing = 1,
    logfile = '/logs/logger.log')
Let’s run it locally!

Note

In theory you could, but this is not something you should run in RStudio.

1. Create a Kinesis Stream
2. Create an IAM user with DynamoDB and Kinesis permissions
3. Write data to the Stream
4. Run the MultiLangDaemon referencing the properties file
Let's run it locally!

Note
In theory you could, but this is not something you should run in RStudio.

1. Create a Kinesis Stream
2. Create an IAM user with DynamoDB and Kinesis permissions
3. Write data to the Stream
4. Run the MultiLangDaemon referencing the properties file
Create a Kinesis Stream

Amazon Kinesis

Amazon Kinesis services make it easier to work with real-time streaming data in the AWS Cloud.

Amazon Kinesis Firehose
Continuously deliver streaming data to Amazon S3, Amazon Redshift, and Amazon Elasticsearch Service.

Go to Firehose
Learn more about Firehose

Amazon Kinesis Analytics
Analyze streaming data from Amazon Kinesis Firehose and Amazon Kinesis Streams in real-time using SQL.

Go to Analytics
Learn more about Analytics

Amazon Kinesis Streams
Collect and stream data for ordered, replayable, real-time processing.

Go to Streams
Learn more about Streams

Amazon Kinesis documentation and support

Firehose documentation | Analytics documentation | Streams documentation | Forums
Create a Kinesis Stream

Stream processing using AWR

gitlab.com/cardcorp/AWR
Check the Kinesis Stream
Create an IAM user
library(rJava)
.jcall("java/lang/System", "S", "setProperty", "aws.profile", "personal")

library(AWR.Kinesis)
library(jsonlite)
library(futile.logger)
library(nycflights13)

while (TRUE) {
  ## pick a ~car~flight
  flight <- flights[sample(1:nrow(flights), 1), ]

  ## prr <- .jnew('com.amazonaws.services.kinesis.model.PutRecordRequest')
  ## prr$setStreamName('test1')
  ## prr$setData(J('java.nio.ByteBuffer')$wrap(.jbyte(charToRaw(toJSON(car)))))
  ## prr$setPartitionKey(rownames(car))
  ## kc$putRecord(prr)

  res <- kinesis_put_record(stream = 'test-AWR', region = 'us-east-1',
                            data = toJSON(flight), partitionKey = flight$dest)
  flog.info(paste('Pushed a new flight to Kinesis:', res$sequenceNumber))
}

Gergely Daroczi (@daroczig)
Write Data to the Stream from R

Gergely Daroczi (@daroczig)

Stream processing using AWR

gitlab.com/cardcorp/AWR
library(futile.logger)
library(AWR.Kinesis)

kinesis_consumer(

initialize = function()
  flog.info('Hello'),

processRecords = function(records)
  flog.info(paste('Received', nrow(records), 'records from Kinesis')),

shutdown = function()
  flog.info('Bye'),

updater = list( 
  list(1, function()
    flog.info('Updating some data every minute')),
  list(1/60*10, function()
    flog.info(paste( 
      'This is a high frequency updater call',
      'running every 10 seconds'))),

checkpointing = 1,
logfile = '/logs/logger.log')
Running the MultiLangDaemon locally

Gergely Daroczi (@daroczig)

Stream processing using AWR

gitlab.com/cardcorp/AWR
Running the MultiLangDaemon locally

Gergely Daroczi (@daroczig)

Stream processing using AWR

gitlab.com/cardcorp/AWR
Let’s run it in AWS!

1. Dockerize your Kinesis Consumer:
   - Java
   - R
   - AWR, AWR.Kinesis packages
   - app.R
   - app.properties
   - startup command

2. Put it on Docker Hub

3. Run as a EC2 Container Service Task:
   - Create an ECS cluster
   - Create ECS Task Role
   - Create a Task definition
   - Run it (as a service)
Dockerize your Kinesis Consumer

Gergely Daroczi (@daroczig)

Stream processing using AWR

gitlab.com/cardcorp/AWR
Dockerize your Kinesis Consumer

```
FROM cardcorp/r-kinesis:latest
MAINTAINER Gergely Daroczi <gergely.daroczi@card.com>
## Add consumer
COPY files /app
```

Gergely Daroczi (@daroczig)
Dockerize your Kinesis Consumer

Gergely Daroczi (@daroczig)

Stream processing using AWR

gitlab.com/cardcorp/AWR
Dockerize your Kinesis Consumer

Gergely Daroczi (@daroczig)

Stream processing using AWR
Put it on Docker Hub
Create an ECS cluster

Amazon ECS
- Clusters
- Task Definitions
- Repositories

Create Cluster

When you run tasks using Amazon ECS, you place them on a cluster, which is a logical grouping of EC2 instances. This wizard will guide you through the process to create a cluster. You will name your cluster, and then configure the container instances that your tasks can be placed on, the security group for your container instances to use, and the IAM role to associate with your container instances so that they can make calls to the AWS APIs on your behalf.

Cluster name: AWR-test
- Create an empty cluster

EC2 Instance type: t2.medium

Number of instances: 1

EC2 Ami Id: ami-ami-2016.09.1-amazon-ecs-optimized-amz-b2df2ca4

EBS storage (GiB): 22

Key pair: You will not be able to SSH into your EC2 instances without a key pair. You can create a new key pair in the EC2 console.

Networking

Configure the VPC for your container instances to use. A VPC is an isolated portion of the AWS cloud populated by AWS objects, such as Amazon EC2 instances. You can choose an existing VPC, or create a new one with this wizard.

VPC: Create a new vpc

CIDR Block: 10.0.0.0/16
Create ECS Task Role

Set Role Name

Enter a role name. You cannot edit the role name after the role is created.

Role Name: AWR-ECS

Maximum 64 characters. Use alphanumeric and "_" characters.
Create ECS Task Role

Select Role Type

AWS Service Roles

- Amazon EC2 Role for EC2 Container Service
  Role to allow EC2 instances in an Amazon ECS cluster to access Amazon ECS.
- Amazon EC2 Container Service Role
  Allows ECS to create and manage AWS resources on your behalf.
- Amazon EC2 Container Service Task Role
  Allows ECS tasks to call AWS services on your behalf.
- Amazon EC2 Spot Fleet Role
  Role to allow EC2 Spot Fleet to request and terminate Spot instances on your behalf.
- Amazon Elastic MapReduce
  Role to allow EMR to access other AWS services such as EC2 on your behalf.

Role for Cross-Account Access

Role for Identity Provider Access
Create ECS Task Role

Step 1: Set Role Name
Step 2: Select Role Type
Step 3: Establish Trust
Step 4: Attach Policy
Step 5: Review

Attach Policy

Select one or more policies to attach. Each role can have up to 10 policies attached.

Filter: Policy Type

<table>
<thead>
<tr>
<th>Policy Name</th>
<th>Attached Entities</th>
<th>Creation Time</th>
<th>Edited Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>AmazonKinesisFullAccess</td>
<td>1</td>
<td>2015-02-06 10:40 PST</td>
<td>2015-02-06 10:40 PST</td>
</tr>
<tr>
<td>AmazonKinesisReadOnlyA...</td>
<td>1</td>
<td>2015-02-06 10:40 PST</td>
<td>2015-02-06 10:40 PST</td>
</tr>
<tr>
<td>AmazonKinesisAnalyticsF...</td>
<td>0</td>
<td>2016-09-21 12:01 PST</td>
<td>2016-09-21 12:01 PST</td>
</tr>
<tr>
<td>AmazonKinesisAnalyticsR...</td>
<td>0</td>
<td>2016-09-21 11:16 PST</td>
<td>2016-09-21 11:16 PST</td>
</tr>
<tr>
<td>AmazonKinesisFirehoseF...</td>
<td>0</td>
<td>2015-10-07 11:45 PST</td>
<td>2015-10-07 11:45 PST</td>
</tr>
<tr>
<td>AmazonKinesisFirehoseR...</td>
<td>0</td>
<td>2015-10-07 11:43 PST</td>
<td>2015-10-07 11:43 PST</td>
</tr>
<tr>
<td>AWSLambdaKinesisExecute...</td>
<td>0</td>
<td>2015-04-09 08:14 PST</td>
<td>2015-04-09 08:14 PST</td>
</tr>
</tbody>
</table>

Cancel  Previous  Next Step
Create a Task definition

A task definition specifies which containers are included in your task and how they interact with each other. You can also specify data volumes for your containers to use.

- **Task Definition Name**: AWR-logger
- **Task Role**: AWR-ECS
- **Network Mode**: Bridge

**Constraints**
Constraints allow you to filter the instances used for your placement strategies using built-in or custom attributes. The scheduler first filters the instances that match the constraints and then applies the placement strategy to place the task.

**Container Definitions**
No results

**Volumes**
No results

[Image of the Amazon ECS console showing the Create a Task Definition page with filled out fields: Task Definition Name as AWR-logger, Task Role as AWR-ECS, Network Mode as Bridge, and details of Container Definitions and Volumes sections.]
Create a Task definition

Add volume

Name: logs
Source path: /logs

Constraint

Container Definitions

Add container

Container Name | Image | Hard/Soft memory limits (MB) | Essential
---|---|---|---

Volumes

Name | Source Path
---|---

Add volume

Configure via JSON
Create a Task definition
Create a Task definition

Gergely Daroczi (@daroczig)

Stream processing using AWR

gitlab.com/cardcorp/AWR
Run the ECS Task

Amazon EC2 Console

Task Definition Name: AWR-logger

Status: Active

Create new revision
Run Task
Create Service
Update Service
Deregister

AWS Console: https://console.aws.amazon.com/ecs/home?region=us-east-1#/taskDefinitions/AWR-logger/status ACTIVE
Run the ECS Task

https://console.aws.amazon.com/ecs/home?region=us-east-1#/clusters/AWR/tasks

Cluster: AWR

Status: ACTIVE
- Registered container instances: 1
- Pending tasks count: 0
- Running tasks count: 1

Services > Tasks

Desired task status: Running/Stopped

Task: Oc9f224a-780... Task Definition: AWR-logger:1 Group: family:AWR-log... Container Instance: 2704936-70e2... Status: RUNNING Desired status: RUNNING

Last updated on March 5, 2017 6:05:01 PM (0m ago)
Run the ECS Task

Gergely Daroczi (@daroczig)

Stream processing using AWR

gitlab.com/cardcorp/AWR
Scaling the Kinesis Consumer up

Gergely Daroczi (@daroczig)

Stream processing using AWR

gitlab.com/cardcorp/AWR
Nice example project, but . . .

- I might want to avoid publishing my Consumer on Docker Hub
- I might want to avoid publishing my code on GitHub
- I might want to avoid committing credentials etc to the repo

Problems:

- How to store credentials in the Docker images?
- Where to store the Docker images?
Nice example project, but...

- I might want to avoid publishing my Consumer on Docker Hub
- I might want to avoid publishing my code on GitHub
- I might want to avoid committing credentials etc to the repo

Problems:

- How to store credentials in the Docker images? KMS
- Where to store the Docker images? ECR
KMS

Source: AWS Encryption SDK
Current AWR.KMS Features

- encrypt up to 4 KB of arbitrary data:

```r
> library(AWR.KMS)
> kms_encrypt('alias/mykey', 'foobar')
[1] "Base-64 encoded ciphertext"
```

- decrypt such Base-64 encoded ciphertext back to plaintext:

```r
> kms_encrypt('Base-64 encoded ciphertext')
[1] "foobar"
```

- generate a data encryption key:

```r
> kms_generate_data_key('alias/mykey')
$cipher
[1] "Base-64 encoded, encrypted data encryption key"
$key
[1] "alias/mykey"
$text
[1] 00 01 10 11 00 01 10 11 ...
```
let's say we want to encrypt the mtcars dataset stored in JSON

```r
library(jsonlite)
data <- toJSON(mtcars)
```

generate a 256-bit data encryption key (that's supported by digest::AES)

```r
library(AWR.KMS)
key <- kms_generate_data_key(‘alias/mykey’, byte = 32L)
```

convert the JSON to raw so that we can use that with digest::AES

```r
raw <- charToRaw(data)
```

the text length must be a multiple of 16 bytes

```r
raw <- c(raw, as.raw(rep(0, 16 - length(raw) %% 16)))
```

encrypt the raw object with the new key + digest::AES

```r
library(digest)
aes <- AES(key$text)
base64_enc(aes$encrypt(raw))
```

decrypt the above returned ciphertext using the decrypted key

```r
rawToChar(aes$decrypt(base64_dec(...), raw = TRUE))
```
Example “Production” Consumer App

library(AWR.Kinesis); library(jsonlite); library(AWR.KMS); library(futile.logger); flog.threshold(DEBUG)

kinesis_consumer(
    initialize = function() {
        flog.info('Decrypting Redis hostname via KMS')
        host <- kms_decrypt('AQECAHiiz4GEFPFQLL9AA0N5TY/1DR5euQQScpXQU9iYTn+u...')
        flog.info('Connecting to Redis')
        library(rredis); redisConnect(host = host)
        flog.info('Connected to Redis')
    },
    processRecords = function(records) {
        flog.info(paste('Received', nrow(records), 'records from Kinesis'))
        for (record in records$data) {
            flight <- fromJSON(record)$dest
            if (!is.null(flight)) {
                flog.debug(paste('Adding +1 to', flight))
                redisIncr(sprintf('flight:%s', flight))
            } else {
                flog.error('Flight destination not found')
            }
        }
    },
    updater = list(
        list(1/6, function() {
            flog.info('Checking overall counters')
            flights <- redisKeys('flight:*')
            for (flight in flights) {
                flog.debug(paste('Found', redisGet(flight), sub('^flight:', '', flight)))
            }
        })),
    logfile = '/logs/redis.log')
Dockerfile:

FROM cardcorp/r-kinesis:latest
MAINTAINER Gergely Daroczi <gergely.daroczi@card.com>

## Install R package to interact with Redis
RUN install2.r --error rredis && rm -rf /tmp/downloaded_packages/ /tmp/*.rds

## Add consumer
COPY files /app

Build and push to ECR:

docker build -t cardcorp/r-kinesis-secret .
`aws ecr get-login --region us-east-1`
docker tag -f cardcorp/r-kinesis-secret:latest \\   ***.dkr.ecr.us-east-1.amazonaws.com/cardcorp/r-kinesis-secret:latest
docker push ***.dkr.ecr.us-east-1.amazonaws.com/cardcorp/r-kinesis-secret:latest
library(treemap); library(highcharter); library(nycflights13)
library(rredis); redisConnect(host = '***', port = '***')

ui <- shinyUI(highchartOutput('treemap', height = '800px'))
server <- shinyServer(function(input, output, session) {

  destinations <- reactive({
    reactiveTimer(2000)()
    flights <- redisMGet(redisKeys('flight:*'))
    flights <- data.frame(faa = sub('^flight:', '', names(flights)),
                          N = as.numeric(flights))
    merge(flights, airports, by = 'faa')
  })

  output$treemap <- renderHighchart({
    tm <- treemap(destinations(), index = c('faa'),
                  vSize = 'N', vColor = 'tz',
                  type = 'value', draw = FALSE)
    hc_title(hctreemap(tm, animation = FALSE), text = 'Flights from NYC')
  })
})

shinyApp(ui = ui, server = server)
Technical Details

- AWR repo:
  - 9.7 GB
  - 388 tags/versions
  - GitLab + CI + drat

install.packages('AWR', repos = 'https://cardcorp.gitlab.io/AWR')
Technical Details

- **AWR repo:**
  - 9.7 GB
  - 388 tags/versions
  - GitLab + CI + drat

```r
install.packages('AWR', repos = 'https://cardcorp.gitlab.io/AWR')
```

- Submitted to CRAN on
  - 2016-12-05
Technical Details

- AWR repo:
  - 9.7 GB
  - 388 tags/versions
  - GitLab + CI + drat

```
install.packages('AWR', repos = 'https://cardcorp.gitlab.io/AWR')
```

- Submitted to CRAN on
  - 2016-12-05
  - 2017-01-09
  - 2017-01-10
  - 2017-01-11
  - 2017-01-11
  - 2017-01-13
Technical Details

- AWR repo:
  - 9.7 GB
  - 388 tags/versions
  - GitLab + CI + drat

```r
install.packages('AWR', repos = 'https://cardcorp.gitlab.io/AWR')
```

- Submitted to CRAN on
  - 2016-12-05
  - 2017-01-09
  - 2017-01-10
  - 2017-01-11
  - 2017-01-11
  - 2017-01-13

- Release cycle: 2 minor, ~125 patch versions in the past 12 months
- CI
library(rJava)
kc <- .jnew('com.amazonaws.services.s3.AmazonS3Client')
kcs$getS3AccountOwner()$getDisplayName()
[1] "foobar"
Because "S" is so 1992.