Gordon Documentation

Release 0.7.0

Jorge Bastida

Dec 14, 2017

Contents

Welcome to Gordon's documentation. I recommend that you get started with *Installation* and then head over to the *Quickstart*. Besides the Quickstart, there are also several tutorials for some of the different available event sources.

If you are the kind of person who wants to learn by example, there are lot's in our Examples directory in github.

CHAPTER 1

First Steps

Your first steps using gordon.

1.1 Installation

Gordon requires several python libraries, but all of them should get installed seamlessly using `pip`.

1.1.1 Using pip

```
$ pip install gordon
```

If you are on OSX El Capitan, use the following (Why? Check Donald Stufft's comment in pypa/pip)

```
$ pip install gordon --ignore-installed six
```

1.1.2 Development version

The source code of Gordon is available on Github https://github.com/jorgebastida/gordon/.

You can install this version using:

```
$ python setup.py develop
```

1.1.3 What next?

Give it a look to the *Quickstart* where you'll create your first Gordon project!

1.2 Quickstart

In which language you feel more comfortable?

1.2.1 Quickstart: Python

Now that you have Gordon installed, let's create our first project. Before doing so, you need to understand how Gordon projects are structured.

A Gordon project consist in one or more applications. The term application describes a directory that provides some set of features. Applications may be reused in various projects.

Requirements for python lambdas:

- `pip`: https://pypi.python.org/pypi/pip

Creating a project

From the command line, cd into a directory where you'd like to store your code, then run the following command:

```
$ gordon startproject demo
```

This will create a *demo* directory in your current directory with the following structure:

```
demo
- settings.yml
```

As you can imagine that *settings.yml* file will contain most of our project-wide settings. If you want to know more, *Settings* will tell you how the settings work.

Creating an application

Now that we have our project created, we need to create our first app. Run the following command from the command line:

```
$ gordon startapp firstapp
```

Note: You can create lambdas in any of the AWS supported languages (Python, Javascript and Java) and you can mix them within the same project and app. By default `startapp` uses the python runtime, but you can pick a different one by adding `--runtime=js|java` to it.

This will create a *firstapp* directory inside your project with the following structure:

```
firstapp/
- helloworld
|    - code.py
- settings.yml
```

These files are:

- `code.py` : File where the source code of our first `helloworld` lambda will be. By default gordon creates a function called `handler` inside this file and registers it as the main handler.

- `settings.yml` : Configuration related to this application. By default gordon registers a `helloworld` lambda function.

Once you understands how everything works, and you start developing your app, you'll rename/remove this function, but to start with we think this is the easiest way for you to understand how everything works.

Give it a look to `firstapp/settings.yml` and `firstapp/helloworld/code.py` files in order to get a better understanding of what gordon just created for you.

Now that we know what these files does, we need to install this `firstapp`. In order to do so, open your project `settings.yml` and add `firstapp` to the `apps` list:

```
---
project: demo
default-region: us-east-1
code-bucket: gordon-demo-5f1fb41f
apps:
  - gordon.contrib.lambdas
  - firstapp
```

This will make Gordon take count of the resources registered within the `firstapp` application.

Build your project

Now that your project is ready, you need to build it. You'll need to repeat this step every time you make some local changes and want to deploy them to AWS.

From the command line, cd into the project root, then run the following command:

```
$ gordon build
```

This command will have an output similar to:

```
$ gordon build
Loading project resources
Loading installed applications
  contrib_lambdas:
    ✓ version
  firstapp:
    ✓ helloworld
Building project...
  ✓ 0001_p.json
    ✓ lambda:contrib_lambdas:version
    ✓ lambda:firstapp:helloworld
  ✓ 0002_pr_r.json
  ✓ 0003_r.json
```

What is all this? Well, without going into much detail, gordon has just decided that deploying you application implies three stag

- `0001_p.json` gordon is going to create a s3 bucket where the code of your lambdas will be uploaded.
- `0002_pr_r.json` gordon will upload the code of your lambdas to S3.
- `0003_r.json` gordon will create your lambdas.

But, should I care? **No** you should not really care much at this moment about what is going on. The only important part is that you'll now see a new `_build` directory in your project path. That directory contains everything gordon needs to put your lambdas live.

If you want to read more about the internals of gordon project, you read more in the *Project* page.

Deploy your project

Deploying a project is a as easy as using the `apply` command:

```
$ gordon apply
```

Note: It is important that you make your AWS credential available in your terminal before, so gordon can use them. For more information: *Setup AWS Credentials*

This command will have an output similar to:

```
$ gordon apply
Applying project...
  0001_p.json (cloudformation)
    CREATE_COMPLETE waiting...
  0002_pr_r.json (custom)
    ✓ code/contrib_lambdas_version.zip (da0684c2)
    ✓ code/firstapp_helloworld.zip (45da7d76)
  0003_r.json (cloudformation)
    CREATE_COMPLETE waiting...
```

Your lambdas are ready to be used! Navigate to AWS: Lambdas to test them.

What next?

You should have a basic understanding of how Gordon works. We recommend you to dig a bit deeper and explore:

- *Project* Details about how you can customize your projects
- *Lambdas* In-depth exmplanation of how lambdas work.
- *Event Sources* List of all resources and integrations you can create using Gordon.

1.2.2 Quickstart: Javascript

Now that you have Gordon installed, let's create our first project. Before doing so, you need to understand how Gordon projects are structured.

A Gordon project consist in one or more applications. The term application describes a directory that provides some set of features. Applications may be reused in various projects.

Requirements for Javascript lambdas:

- `npm`: https://nodejs.org/en/download/

Creating a project

From the command line, cd into a directory where you'd like to store your code, then run the following command:

```
$ gordon startproject demo
```

This will create a *demo* directory in your current directory with the following structure:

```
demo
- settings.yml
```

As you can imagine that *settings.yml* file will contain most of our project-wide settings. If you want to know more, *Settings* will tell you how the settings work.

Creating an application

Now that we have our project created, we need to create our first app. Run the following command from the command line:

```
$ gordon startapp firstapp --runtime=js
```

Note: You can create lambdas in any of the AWS supported languages (Python, Javascript and Java) and you can mix them within the same project and app. By default `startapp` uses the python runtime, but you can pick a different one by adding `--runtime=py|java` to it.

This will create a *firstapp* directory inside your project with the following structure:

```
firstapp/
- helloworld
|    - code.js
- settings.yml
```

Note: If you pick `python` or `java` as runtime, the layout will not be 100% the same, but pretty similar.

These files are:

- `code.js` : File where the source code of our first `helloworld` lambda will be. By default gordon creates a function called `handler` inside this file and registers it as the main handler.
- `settings.yml` : Configuration related to this application. By default gordon registers a `helloworld` lambda function.

Once you understand how everything works, and you start developing your app, you'll rename/remove this function, but to start with we think this is the easiest way for you to understand how everything works.

Give it a look to `firstapp/settings.yml` and `firstapp/helloworld/code.js` files in order to get a better understanding of what gordon just created for you.

Now that we know what these files does, we need to install this `firstapp`. In order to do so, open your project `settings.yml` and add `firstapp` to the `apps` list:

```
---
project: demo
default-region: us-east-1
code-bucket: gordon-demo-5f1fb41f
apps:
  - gordon.contrib.lambdas
  - firstapp
```

This will make Gordon take count of the resources registered within the `firstapp` application.

Build your project

Now that your project is ready, you need to build it. You'll need to repeat this step every single time you make some local changes and want to deploy them to AWS.

From the command line, cd into the project root, then run the following command:

```
$ gordon build
```

This command will have an output similar to:

```
$ gordon build
Loading project resources
Loading installed applications
  contrib_lambdas:
    ✓ version
  firstapp:
    ✓ helloworld
Building project...
  ✓ 0001_p.json
    ✓ lambda:contrib_lambdas:version
    ✓ lambda:firstapp:helloworld
  ✓ 0002_pr_r.json
  ✓ 0003_r.json
```

What is all this? Well, without going into much detail, gordon has just decided that deploying you application implies three stag

- `0001_p.json` gordon is going to create a s3 bucket where the code of your lambdas will be uploaded.
- `0002_pr_r.json` gordon will upload the code of your lambdas to S3.
- `0003_r.json` gordon will create your lambdas.

But, should I care? **No** you should not really care much at this moment about what is going on. The only important part is that you'll now see a new `_build` directory in your project path. That directory contains everything gordon needs to put your lambdas live.

If you want to read more about the internals of gordon project, you read more in the *Project* page.

Deploy your project

Deploying a project is a as easy as using the `apply` command:

```
$ gordon apply
```

Note: It is important that you make your AWS credential available in your terminal before, so gordon can use them. For more information: *Setup AWS Credentials*

This command will have an output similar to:

```
$ gordon apply
Applying project...
  0001_p.json (cloudformation)
    CREATE_COMPLETE waiting...
  0002_pr_r.json (custom)
    ✓ code/contrib_lambdas_version.zip (da0684c2)
```

```
   ✓ code/firstapp_helloworld.zip (45da7d76)
  0003_r.json (cloudformation)
    CREATE_COMPLETE waiting...
```

Your lambdas are ready to be used! Navigate to AWS: Lambdas to test them.

What next?

You should have a basic understanding of how Gordon works. We recommend you to dig a bit deeper and explore:

- *Project* Details about how you can customize your projects
- *Lambdas* In-depth exmplanation of how lambdas work.
- *Event Sources* List of all resources and integrations you can create using Gordon.

1.2.3 Quickstart: Java

Now that you have Gordon installed, let's create our first project. Before doing so, you need to understand how Gordon projects are structured.

A Gordon project consists of one or more applications. The term application describes a directory that provides some set of features. Applications may be reused in various projects.

Requirements for Java lambdas:

- `gradle`: http://gradle.org/gradle-download/

Creating a project

From the command line, cd into a directory where you'd like to store your code, then run the following command:

```
$ gordon startproject demo
```

This will create a *demo* directory in your current directory with the following structure:

```
demo
- settings.yml
```

As you can imagine that *settings.yml* file will contain most of our project-wide settings. If you want to know more, *Settings* will tell you how the settings work.

Creating an application

Now that we have our project created, we need to create our first app. Run the following command from the command line:

```
$ gordon startapp firstapp --runtime=java
```

Note: You can create lambdas in any of the AWS supported languages (Python, Javascript and Java) and you can mix them within the same project and app. By default `startapp` uses the python runtime, but you can pick a different one by adding `--runtime=py|javascript` to it.

This will create a *firstapp* directory inside your project with the following structure:

```
firstapp/
- helloworld
|   - build.gradle
|   - src
|       - main
|           - java
|               - helloworld
|                   - Hello.java
- settings.yml
```

Note: If you pick `python` or `js` as runtime, the layout will not be 100% the same, but pretty similar.

These files are:

- `helloworld.java` : File where the source code of our first helloworld lambda will be. By default gordon creates a function called `handler` in this file.
- `build.gradle` : Gradle file gordon will use to build your lambda.
- `settings.yml` : Configuration related to this application. By default gordon registers a `helloworld` lambda the function within `Hello.java`.

Once you understand how everything works, and you start developing your app, you'll rename/remove this function, but to start with we think this is the easiest way for you to understand how everything works.

Give it a look to `firstapp/settings.yml` and `firstapp/helloworld/src/main/java/helloworld/Hello.java` files in order to get a better understanding of what gordon just created for you.

Now that we know what these files does, we need to install this `firstapp`. In order to do so, open your project `settings.yml` and add `firstapp` to the `apps` list:

```
---
project: demo
default-region: us-east-1
code-bucket: gordon-demo-5f1fb41f
apps:
  - gordon.contrib.lambdas
  - firstapp
```

This will make Gordon take count of the resources registered within the `firstapp` application.

Build your project

Now that your project is ready, you need to build it. You'll need to repeat this step every single time you make some local changes and want to deploy them to AWS.

From the command line, cd into the project root, then run the following command:

```
$ gordon build
```

This command will have an output similar to:

```
$ gordon build
Loading project resources
```

```
Loading installed applications
  contrib_lambdas:
    ✓ version
  firstapp:
    ✓ helloworld
Building project...
  ✓ 0001_p.json
  ✓ 0002_pr_r.json
  ✓ 0003_r.json
```

What is all this? Well, without going into much detail, gordon has just decided that deploying you application implies three stag

- `0001_p.json` gordon is going to create a s3 bucket where the code of your lambdas will be uploaded.
- `0002_pr_r.json` gordon will upload the code of your lambdas to S3.
- `0003_r.json` gordon will create your lambdas.

But, should I care? **No** you should not really care much at this moment about what is going on. The only important part is that you'll now see a new `_build` directory in your project path. That directory contains everything gordon needs to put your lambdas live.

If you want to read more about the internals of gordon project, you read more in the *Project* page.

Deploy your project

Deploying a project is a as easy as using the `apply` command:

```
$ gordon apply
```

Note: It is important that you make your AWS credential available in your terminal before, so gordon can use them. For more information: *Setup AWS Credentials*

This command will have an output similar to:

```
$ gordon apply
Applying project...
0001_p.json (cloudformation)
  CREATE_COMPLETE waiting... -
0002_pr_r.json (custom)
  ✓ code/contrib_lambdas_version.zip (c3137e97)
  ✓ code/firstapp_helloworld.zip (c7ec05a8)
0003_r.json (cloudformation)
  CREATE_COMPLETE
```

Your lambdas are ready to be used! Navigate to AWS: Lambdas to test them.

What next?

You should have a basic understanding of how Gordon works. We recommend you to dig a bit deeper and explore:

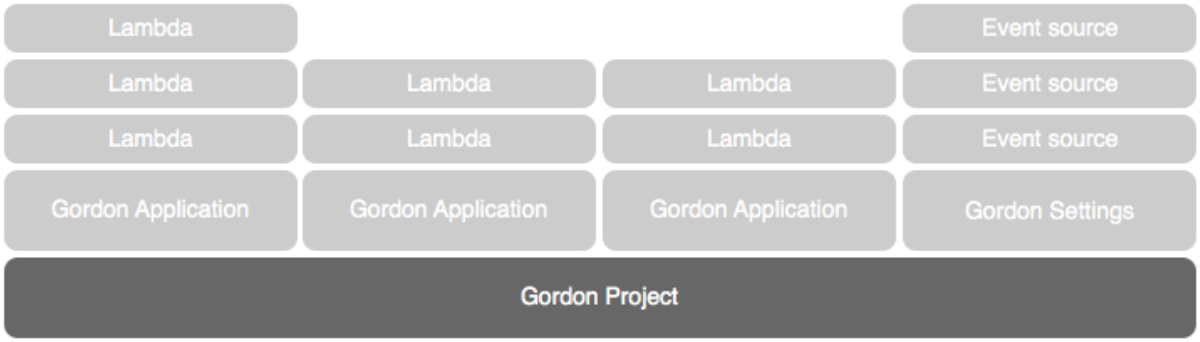
- *Project* Details about how you can customize your projects
- *Lambdas* In-depth explanation of how lambdas work.

- *Event Sources* List of all resources and integrations you can create using Gordon.

CHAPTER 2

Documentation

2.1 Project



Projects are the root container which contain all other resources. Projects are composed of:

- apps
- settings

Once you start using gordon, you'll realize it makes sense to have several projects, and not only one monolitic one with dozens of applications.

This is not news to you if you have heard about microservices, but it is good to emphasize that massive projects might not a good idea (generally speaking).

As always, give it a shoot and decide what is better for you.

2.1.1 How can I create a new project?

Creating a new project is easy, you only need to run the following command:

```
$ gordon startproject demo
```

This will create a new directory called `demo` which will contain the most basic project. That's a single `settings.yml` file:

```
demo
- settings.yml
```

2.1.2 Project Actions

Once you have created your project, there are two main actions that you'll run from the command line; `build` and `apply`.

build

Build is the action that will collect all registered resources in your project, and create several templates in the `_build` directory which will represent everything you have defined. As well as creating these templates, gordon will create all necessary artifacts that it'll later upload to s3.

At this point, gordon will not use any AWS credentials. This is important.

What gordon generates in the build directory is merely declarative and completely agnostic of in which region or stage you'll (later) deploy it. Gordon doesn't know/care what is the current status (if any) of all those resources.

This is one of the greatness (and technical challenges) of gordon.

The number of required templates depend on you project, but these are all possible templates gordon will create:

Acronym	Name	Description
`pr_p.json`	Pre Project	Custom template - This is not generally used
`p.json`	Project	CloudFormation template - Gordon will create a S3 bucket where it'll upload your lambdas
`pr_r.json`	Pre Resources	Custom template - Gordon will generally upload your lambdas to S3.
`r.json`	Resources	CloudFormation template - Gordon will create your lambdas and event sources
`ps_r.json`	Post Resources	Custom template - This is not generally used

apply

Apply is the action that will deploy your project to one specific region and stage.

Term	Description
`region`	AWS cloud is divided in several regions. Each Region is a separate geographic area. AWS Regions and Availability Zones
`stage`	Stages are 100% isolated deployments of the same project. The idea is that the same project can be deployed in the same AWS account in different stages (`dev`, `staging`, `prod`...) in order to SAFELY test it's behaviour. Stage names must only contain up to 16 lowercase alphanumeric characters including hyphen.

This command will:

- Collect all required parameters for this stage.

- Sequentially apply all gordon templates.

This command (for obvious reasons), will use your AWS credentials to apply your project templates.

delete

Removes all deployed project resources of one specific region and stage.

This is a **destructive action**, so gordon will by default do a `dry-run` and output all resources which would be deleted.

If you are ok with those resources being deleted, you can run the same command but adding the argument `--confirm` in order to confirm your desire of gordon deleting all of them.

2.1.3 Anatomy of the project

```
---
project: { STRING }
default-region: { AWS_REGION }
code-bucket: { STRING }
apps:
  - { STRING }
vpc: { MAP }
contexts: { MAP }
```

2.1.4 Lambda Properties

Project Name

Name	`project`
Required	Yes
Valid types	`string`
Description	Name for your Project

default-region

Name	`default-region`
Required	Yes
Valid types	`string`
Description	Default region where the project will be deployed

code-bucket

Name	`code-bucket`
Required	Yes
Valid types	`string`
Validation	Up to 31 lowercase alphanumeric characters including hyphen.
Description	Base Name of the bucket gordon will use to store the source code of your lambdas and Cloudformation templates.

Because the source code and the lambdas needs to be in the same region, gordon will create on bucket per region and stage following the following format:

`$CODE_BUCKET-$REGION-$STAGE`.

apps

Name	`apps`
Required	Yes
Valid types	`list`
Description	List of installed apps

By default when you create a project, gordon will include some applications which you'll probably need. Those applications are called *gordon.contrib* applications and provide you (and your gordon project) with some basic functionalities that you (or gordon) might need.

vpc

Name	`vpc`
Required	No
Valid types	`map`
Description	Map of vpc names with their respective `security-groups` and `subnet-ids`.

For more information *Lambdas vpc setting*.

Example:

```
---
project: vpcexample
...

vpcs:
    my-vpc:
        security-groups:
            - sg-00000000
        subnet-ids:
            - subnet-1234567a
            - subnet-1234567b
            - subnet-1234567c
```

You can customize both `security-groups` and `subnet-ids` using parameters

```
---
project: vpcexample
...

vpcs:
    my-vpc:
        security-groups: ref://VpcSecurityGroups
        subnet-ids: ref://VpcSubnets
```

contexts

Name	`contexts`
Required	No
Valid types	`map`
Description	Map of context names with their definitions.

For more information *Contexts*.

Example:

```
---
project: example
...

contexts:
  default:
    database_host: 10.0.0.1
    database_username: dev-bob
    database_password: shrug
```

2.2 Lambdas

Gordon has two aims:

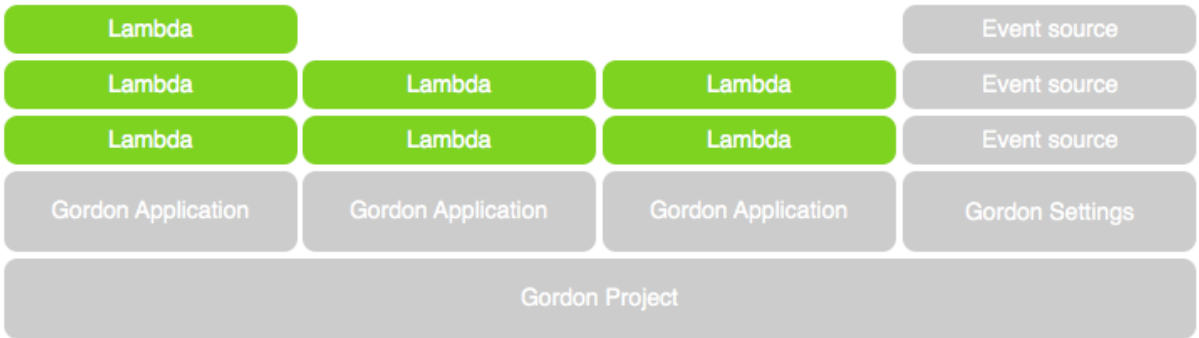
- Easily deploy and manage lambdas.
- Easily connect those lambdas to other AWS services (kinesis, dynamo, s3, etc...)

Lambdas are simple functions written in any of the supported AWS languages (python, javascript and java). If you want to know more, you can read AWS documentation in the topic:

- What Is AWS Lambda?
- AWS Lambda FAQs
- AWS Lambda Limits

Working with lambdas is quite easy to start with, but once you want to develop some complex integrations, it becomes a bit of a burden to deal with all the required steps to put some changes live. Gordon tries to make the entire process as smooth as possible.

In gordon, Lambdas are resources that you'll group and define within apps. The idea is to keep Lambdas with the same business domain close to each other in the same app.



Before we continue, there is a bit of terminology we need to make clear:

Term	Description
`lambda`	Is a static working piece of code ready to be run on AWS.
`lambda version`	Static point-in-time representation of a working lambda.
`lambda alias`	Pointer to a lambda.
`code bucket`	S3 bucket where your lambda code is uploaded.
`code`	S3 Object which contains your lambda code and all required libraries/packages (zip)
`code version`	S3 Object Version of one of your lambda code.
`runtime`	Language in which the lambda code is written (python, javascript or java)

2.2.1 What gordon will do for you?

- Download any external requirements your lambdas might have.
- Create a zip file with your lambda, packages and libraries.
- Upload this file to S3.
- Create a lambda with your code and settings (memory, timeout...)
- Publish a new version of the lambda.
- Create an alias named `current` pointing to this new version.
- Create a new IAM Role for this lambda and attach it.

As result, your lambda will be ready to run on AWS!

As you can imagine, this is quite a lot of things to do every time you want to simply deploy a new change! That's where gordon tries to help.

With simply two commands, `build` and `apply` you'll be able to deploy your changes again and again with no effort.

2.2.2 Why the current alias is important?

The `current` alias gordon creates pointing to your most recent lambda is really important. When gordon creates a new event sources (like S3, Dynamo or Kinesis), it'll make those call the lambda aliased as `current` instead of the `$LATEST`.

This is really important to know, because it enables you to (in case of neccesary) change your `current` alias to point to any previous version of the same lambda without needing to re-configure all related event sources.

Any subsequent deploy to the same stage will point the `current` alias to your latest function.

For more information you can read AWS Lambda Function Versioning and Aliases.

2.2.3 Anatomy of a Lambda

The following is the anatomy of a lambda in gordon.

```
lambdas:

  { LAMBDA_NAME }:
    code: { PATH }
    handler: { STRING }
```

```
memory: { NUMBER }
timeout: { NUMBER }
runtime: { RUNTIME_NAME }
description: { STRING }
build: { STRING }
role: { MAP }
vpc: { STRING }
context: { CONTEXT_NAME }
context-destination: { PATH }
auto-vpc-policy: { BOOLEAN }
auto-run-policy: { BOOLEAN }
cli-output: { BOOLEAN }
environment:
   { MAP }
policies:
  { POLICY_NAME }:
    { MAP }
  ...
```

The best way to organize your lambdas is to register them inside the `settings.yml` file of your apps within your *Project*.

2.2.4 Lambda Properties

Lambda Name

Name	Key of the `lambdas` map.
Required	Yes
Valid types	`string`
Max length	30
Description	Name for your lambda. Try to keep it as short and descriptive as possible.

code

Name	`code`
Required	Yes
Valid types	`string`
Max length	30
Description	Path where the code of your lambda is

When creating lambdas you can:

- **Put all the code of your lambda in the same file and make `code` point to it:**
  - `code: code.py`
  - `code: example.js`
- **Put your code in several files within a folder and make `code` point to this directory:**
  - `code: myfolder`
  - Remember: When you point `code` to a directory you need to remember to specify the `runtime` property of your lambda as gordon can't infer it from the filename.

Simple python lambda:

```
lambdas:
  hello_world:
    code: functions.py
```

Folder javascript lambda:

```
lambdas:
  hello_world:
    code: myfolder
    handler: file.handler
    runtime: nodejs6.10
```

Java lambda:

```
lambdas:
  hello_world:
    code: myfolder
    handler: example.Hello::handler
    runtime: java8
```

handler

Name	`handler`
Required	No
Default	`handler`
Valid types	`string`, `reference`
Max length	30
Description	Name of the function within `code` which will be the entry point of you lambda.

```
lambdas:
  hello_world:
    code: functions.py
    handler: my_handler
```

For lambdas using the java runtime, this handler will need to have the following format (`package.class::method`):

```
lambdas:
  hello_world:
    code: helloworld
    runtime: java8
    handler: helloworld.Hello::handler
```

Note: For more information about Java handlers Java Programming Model Handler Types

memory

Name	memory
Required	No
Default	128
Valid types	integer, reference
Max	1536
Min	128
Description	Amount of memory your lambda will get provisioned with

```
lambdas:
  hello_world:
    code: functions.py
    memory: 1536
```

timeout

Name	timeout
Required	No
Default	3
Valid types	integer, reference
Max	300
Min	1
Description	The function execution time (in seconds) after which Lambda terminates the function

Because the execution time affects cost, set this value based on the function's expected execution time.

```
lambdas:
  hello_world:
    code: functions.py
    timeout: 300
```

runtime

Name	runtime
Required	Depends
Valid types	runtime
Description	Runtime of your lambda

Valid runtimes:

Runtime	AWS Runtime
node, nodejs, node0.10 and nodejs0.10	nodejs
node4.3 and nodejs4.3	nodejs4.3
node6.10`	nodejs6.10
python and python2.7	python2.7
java and java8	java8

If you don't specify any runtime, Gordon tries to auto detect it based on the extensions of the code file.

Extension	AWS Runtime
.js	nodejs6.10
.py	python2.7

For folder based lambdas the code property is a directory and not a file, so the runtime can't be inferred.

For these situations, you can manually specify the runtime using this setting:

```
lambdas:
  hello_world:
    code: hellojava
    runtime: java8
```

description

Name	description
Required	No
Default	*Empty*
Valid types	string, reference
Description	Human-readable description for your lambda.

```
lambdas:
  hello_world:
    code: functions.py
    description: This is a really simple function which says hello
```

build

Name	build
Required	No
Valid types	string, list
Description	Build process for collecting resources of your lambda

This property defines which are the commands gordon needs to run in order to collect all the resources from your lambda and copying them to an empty target directory. Once the collection command finishes, gordon will create a zip file with the content of that folder.

This property has one default implementation per available runtime (Java, Javascript, Python), which covers most of the simple use cases, but there are certain use situations where you might need further fine control.

These are the default implementations gordon will use if you leave this property blank:

Python

```
build:
  - cp -Rf * {target}
  - echo "[install]\nprefix=" > {target}/setup.cfg
  - {pip_path} install -r requirements.txt -q -t {target} {pip_install_extra}
  - cd {target} && find . -name "*.pyc" -delete
```

Node

```
build:
  - cp -Rf * {target}
  - cd {target} && {npm_path} install {npm_install_extra}
```

Java

```
build: {gradle_path} build -Ptarget={target} {gradle_build_extra}
```

As you can see, the value of `build` can be either a string or a list of strings. Gordon will process them sequentially within your lambda directory.

There are certain variables you can use to customize this `build` property.

Variable	Description
`target`	Destination folder where you need to put the code of your lambda
`pip_path`	`pip` path. You can customize this using the `pip-path` setting in your settings
`npm_path`	`npm` path. You can customize this using the `npm-path` setting in your settings
`gradle_path`	`gradle` path. You can customize this using the `gradle-path` setting in your settings
`pip_install_extra`	Extra arguments you can define using `pip-install-extra` in your settings
`npm_install_extra`	Extra arguments you can define using `npm-install-extra` in your settings
`gradle_build_extra`	Extra arguments you can define as part of `gradle-build-extra` in your settings
`project_path`	Root directory of your project
`project_name`	Name of your project
`lambda_name`	Name of your lambda

This is the minimal version of what a build command that copies your lambda directory would look like:

```
lambdas:
  hello_world:
    code: mycode
    runtime: python
    handler: code.handler
    build: cp -Rf * {target}
```

You can use this `build` property in conjunction with some more powerful build tools such as `Makefile`, `npm`, `gulp`, `grunt` or simple `bash` files.

In this example, we make `babel` process our javascript files, and leave them in `TARGET`.

```
lambdas:
  hello_world:
    code: mycode
    runtime: node
    handler: code.handler
    build: TARGET={target} npm run build
```

```
{
  "babel": {
    "presets": [
      "es2015"
    ]
  },
  "devDependencies": {
    "babel-cli": "^6.8.0",
    "babel-preset-es2015": "^6.6.0"
  },
  "scripts": {
      "build": "babel *.js --out-dir $TARGET"
  }
}
```

role

Name	role
Required	No
default	Gordon will create a minimal role for this function
Valid types	arn, reference
Description	ARN of the lambda role this function will use.

If not provided, gordon will create one role for this function and include all necessary policies *(This is the default and most likely behaviour you want).*

```
lambdas:
  hello_world:
    code: functions.py
    role: arn:aws:iam::account-id:role/role-name
```

vpc

Name	vpc
Required	No
Valid types	vpc-name
Description	Name of the vpc where this lambda should be deployed.

If the Lambda function requires access to resources in a VPC, specify a VPC configuration that Lambda uses to set up an elastic network interface (ENI). The ENI enables your function to connect to other resources in your VPC, but it doesn't provide public Internet access.

If your function requires Internet access (for example, to access AWS services that don't have VPC endpoints), configure a Network Address Translation (NAT) instance inside your VPC or use an Amazon Virtual Private Cloud (Amazon VPC) NAT gateway. For more information, see NAT Gateways in the Amazon VPC User Guide.

```
lambdas:
  hello_world:
    code: functions.py
    vpc: my-vpc
```

You need to define some properties about your vpc (in this example my-vpc) in the project settings.

```
---
project: vpcexample
...

vpcs:
    my-vpc:
        security-groups:
            - sg-00000000
        subnet-ids:
            - subnet-1234567a
            - subnet-1234567b
            - subnet-1234567c
```

If auto-vpc-policy is True, gordon will attach to your lambda role the required policy which would allow it to access the vpc. If it is False, you'll need to do this by yourself.

context

Name	`context`
Required	No
default	`default`
Valid types	`context-name`
Description	Name of the context you want to inject into this lambda.

For more information about contexts you can read about them in *Contexts*.

```
lambdas:
  hello_world:
    code: functions.py
    context: context_name
```

context-destination

Name	`context-destination`
Required	No
default	`.context`
Valid types	`string`
Description	Path where gordon should put the context json file.

For more information about contexts you can read about them in *Contexts*.

```
lambdas:
  hello_world:
    code: functions.py
    context-destination: my-customize-context-file.json
```

cli-output

Name	`cli-output`
Required	No
Default	True
Valid types	`boolean`
Description	Output the lambda ARN as part of the `apply` output

environment

Name	`environment`
Required	No
Valid types	`map`
Description	Map of environment variables to attach to this lambda.

policies

Name	`policies`
Required	No
Valid types	`map`
Description	Map of AWS policies to attach to the role of this lambda.

This is the way you'll give permissions to you lambda to connect to other AWS services such as dynamodb, kinesis, s3, etc... For more inforamtion AWS IAM Policy Reference

In the following example we attach one policy called `example_bucket_policy` to our lambda `hello_world` in order to make it possible to read and write a S3 bucket called `EXAMPLE-BUCKET-NAME`.

```
lambdas:
  hello_world:
    code: functions.py
    policies:
      example_bucket_policy:
        Version: "2012-10-17"
        Statement:
          -
            Action:
              - "s3:ListBucket"
              - "s3:GetBucketLocation"
            Resource: "arn:aws:s3:::EXAMPLE-BUCKET-NAME"
            Effect: "Allow"
          -
            Action:
              - "s3:PutObject"
              - "s3:GetObject"
              - "s3:DeleteObject"
              - "dynamodb:GetRecords"
            Resource: "arn:aws:s3:::EXAMPLE-BUCKET-NAME/*"
            Effect: "Allow"
```

auto-vpc-policy

Name	`auto-vpc-policy`
Required	No
Default	True
Valid types	`boolean`
Description	Automatically attach to your lambda enough permissions to get a vpc configured.

If `auto-vpc-policy` is `True`, and you lambda has one `vpc` configured, gordon will attach to your lambda role the required policy which would allow it to access the vpc. If it is `False`, you'll need to do this by yourself.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

auto-run-policy

Name	`auto-run-policy`
Required	No
Default	True
Valid types	`boolean`
Description	Automatically attach to your lambda enough permissions to let it run and push logs to CloudWatch Logs.

If `auto-run-policy` is `True`, gordon will attach to your lambda role the required policy which would allow it to run and push logs.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "arn:aws:logs:*:*:*",
        }
    ]
}
```

2.3 Lambda Requirements

`When in Rome do as the Romans`

We believe developers of lambdas should feel like in home while writing them in their languages. We are not going to come up with a package manager for javascript or a build tool for java better than the existing ones.

For that reason we try to respect as much as possible each runtime *de facto* package managers / build tools.

That is the reason why the default `build` implementation for each kind of runtime uses `pip`, `npm` and `graddle`. If you want to know more about the `build` property of lambdas you can read *Lambda: build*.

Note: For using this functionality, you'll need to make the `code` path for you lambda be a directory. For more information you can read the `code` section in *Lambdas*.

2.3.1 Python requirements

If your python lambda requires some python packages, you can create a `requirements.txt` file in the root of your lambda folder, and gordon will install all those using `pip`.

For more information about the format of this file:

- https://pip.readthedocs.io/en/1.1/requirements.html

Additionally you can customize how gordon invoques `pip` using the following settings:

Setting	Description
`pip-path`	Path to you pip binary Default: `pip`
`pip-install-extra`	Extra arguments you want gordon to use while invoking pip install.

Example `requirements.txt`:

```
requests>=2.0
cfn-response
```

2.3.2 Javascript requirements

If your javascript lambda requires some extra modules, you can create a `package.json` file in the root of your lambda folder, and gordon will invoke `npm install` for you.

For more information:

- https://docs.npmjs.com/files/package.json
- https://docs.npmjs.com/cli/install

Additionally you can customize how gordon invoques `npm` using the following settings:

Setting	Description
`npm-path`	Path to you npm binary Default: `npm`
`npm-install-extra`	Extra arguments you want gordon to use while invoking npm install.

Example `package.json`:

```
{
  "dependencies": {
    "path": "0.11.14"
  }
}
```

2.3.3 Java requirements

If your Java lambda requires some extra packages, you can customize how your Java lambda is built editing your `dependency` section in your `build.gradle` file.

For more information:

- https://docs.gradle.org/current/userguide/dependency_management.html

The only requirement gordon enforces to this `build.gradle` file is that the `build` target leaves whatever you want to get bundled into your lambda in the `dest` folder. You can make this build process as complex as you want/need.

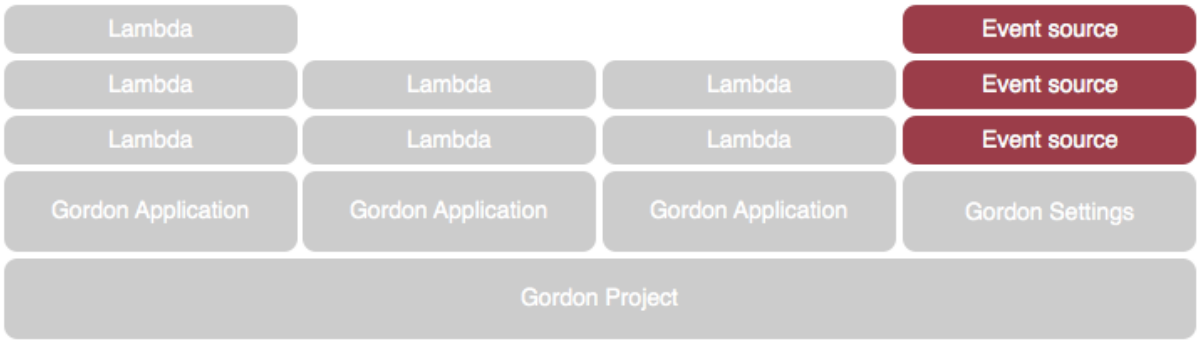
Additionally you can customize how gordon invoques `gradle` using the following settings:

Setting	Description
`gradle-path`	Path to you gradle binary Default: `gradle`
`gradle-build-extra`	Extra arguments you want gordon to use while invoking gradle build.

Example `build.grandle`:

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    compile (
        'com.amazonaws:aws-lambda-java-core:1.1.0',
        'com.amazonaws:aws-lambda-java-events:1.1.0'
    )
}

task buildLambda(type: Copy) {
    from compileJava
    from processResources
    into('lib') {
        from configurations.runtime
    }
    into target
}

build.dependsOn buildLambda
```

2.4 Event Sources



Apart from creating lambdas, gordon can help you wiring your lambdas with lot's of different AWS services. Amazon calls this Event Sources.

2.4.1 Apigateway

Amazon API Gateway is a fully managed service that makes it easy for developers to create APIs at any scale.

Gordon allow you to create and integrate your Lambdas with apigateway resources in order to easily create HTTP APIs.

It might be interesting for you to give it a look to AWS: Amazon API Gateway Concepts before continuing.

Anatomy of the integration

```
apigateway:

  { API_NAME }:
    description: { STRING }
    cli-output: { BOOLEAN }
    resources:
      { URL }:
          methods: { LIST }
          api_key_required: { BOOL }
          authorization_type: { STRING }
          responses: { LIST }
          parameters: { MAP }
          request_templates: { MAP }
          integration:
              type: { STRING }
              lambda: { LAMBDA_NAME }
              http_method: { STRING }
              responses: { LIST }
              parameters: { MAP }
```

Properties

Api Name

Name	Key of the `apigateway` map.
Required	Yes
Valid types	`string`
Max length	30
Description	Name for your apigateway.

Description

Name	`description`
Required	No
Default	*Empty*
Valid types	`string`, `reference`
Max length	30
Description	Description of your api

cli-output

Name	`cli-output`
Required	No
Default	True
Valid types	`boolean`
Description	Output the deployment base URL as part of the `apply` output

Resources

Name	`resources`
Required	Yes
Valid types	`map`
Description	Resources of your API Gateway

Example:

```
apigateway:
    firstapi:
        description: My Inventory
        resources:
            /:
                methods: GET
                lambda: helloworld.index
            /contact/email:
                methods: POST
                lambda: helloworld.contact
```

In this example, we have defined one API called `firstapi` with two resources: `/` and `/contact/email`:

- Each of these urls will call two different lambdas `helloworld.index` and `helloworld.contact` respectively.
- The first url `/` will only allow `GET` requests, and the second one `/contact/email` will only allow `POST` requests.

Resource URL

Name	Key of the `resources` map.
Required	Yes
Valid types	`string`
Description	Full path (url) of your resource

`URLs` are the key of the resources map. For each resource. You need to define the full path including the leading `/`.

If you want to make certain urls have parameters, you can do so using apigatweway syntax.

```
apigateway:
    myshop:
        description: My Inventory API
        resources:
            /:
                methods: GET
                lambda: inventory.index
            /article/{article_id}:
                methods: POST
                lambda: inventory.article
```

Your lambda called `shop.article` will receive one parameter called `article_id`.

Resource Methods

Name	methods
Required	Yes
Valid types	list, string, map
Description	List of valid methods for your resource

Example:

```
apigateway:
    example:
        description: My Api example
        resources:
            /:
                methods: GET
                lambda: inventory.index
            /get_and_post:
                methods: [GET, POST]
                lambda: inventory.article
            /get_post_and_delete:
                methods:
                    - GET
                    - POST
                    - DELETE
                lambda: inventory.article
```

Note: As shortcut, if methods value is a string instead of a list gordon will assume you only want one method.

Resource Methods (advanced)

The simplified version of methods is only a shortcut in order to make gordon's API nicer 95% of the time.

That version (the simplified one) should be more than enough for most of the cases, but if for some reason you want to be able to configure different integrations for each of the methods of an url, you'll need to make methods a map of http methods to integrations.

```
apigateway:
  exampleapi:
    description: My not-that-simple example
    resources:
      /:
        methods:
          GET:
            integration:
              lambda: app.index_on_get
          POST:
            integration:
              lambda: app.index_on_post
```

Note: If you use this approach, you would need to define **ALL** resource settings at the level of each method in your resource.

Resource authorization type

Name	`api_key_required`
Required	No
Default	`False`
Valid Types	`Boolean`
Description	Indicates whether the method requires clients to submit a valid API key.

Name	`authorization_type`
Required	No
Default	`NONE`
Valid Values	`NONE`
Description	Authorization type (if any) for your resource.

Resource Responses

Name	`responses`
Required	No
Valid Types	`Response`
Description	Responses that can be sent to the client who calls this resource.

Example:

```
apigateway:
    helloapi:
        resources:
            /hello:
                method: GET
                integration:
                    lambda: helloworld.sayhi
                    responses:
                        - pattern: ""
                          code: "404"
                responses:
                    - code: "404"
```

Resource Parameters

Name	`parameters`
Required	No
Default	*Empty*
Valid Values	`MAP`
Description	Request parameters that API Gateway accepts. Specify request parameters as key-value pairs (string-to-Boolean maps), with a source as the key and a Boolean as the value. The Boolean specifies whether a parameter is required. A source must match the following format `method.request.$location.$name`, where the `$location` is `querystring`, `path`, or `header`, and `$name` is a valid, unique parameter name.

Resource Request Templates

Name	`request_templates`
Required	No
Valid Values	`map`
Description	A map of Apache Velocity templates that are applied on the request payload. The template that API Gateway uses is based on the value of the Content-Type header sent by the client. The content type value is the key, and the template is the value (specified as a string). For more information: http://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-mapping-template-reference.html

Resource Integration

Name	`integration`
Required	No
Valid Values	`map`
Description	Integration for the current Resource

Integration Type

Name	`type`
Required	No
Default	AWS
Valid Values	`AWS`, `AWS_PROXY`, `MOCK`, `HTTP`
Description	Type of the integration

Integration Lambda

Name	`lambda`
Required	Depends
Valid Values	`app.lambda-name`
Description	Name of the lambda you want to configure for this resource.

Integration Parameters

Name	`parameters`
Required	No
Default	*Empty*
Valid Values	`MAP`
Description	The request parameters that API Gateway sends with the back-end request. Specify request parameters as key-value pairs (string-to-string maps), with a destination as the key and a source as the value. Specify the destination using the following pattern `integration.request.$location.$name`, where `$location` is `querystring`, `path`, or `header`, and `name` is a valid, unique parameter name. The source must be an existing method request parameter or a static value. Static values must be enclosed in single quotation marks and pre-encoded based on their destination in the request.

Integration HTTP Method

Name	`http_method`
Required	Depends
Valid Values	`string`
Description	Http method the ApiGateway will use to contact the integration

Integration Responses

Name	`responses`
Required	No
Valid Values	`list`
Description	The response that API Gateway provides after a method's back end completes processing a request. API Gateway intercepts the integration's response so that you can control how API Gateway surfaces back-end responses. Each response item can conatain the following keys: `pattern`: a regular expression that specifies which error strings or status codes from the back end map to the integration response; `code`: the status code that API Gateway uses to map the integration response to a MethodResponse status code; `template`: the templates used to transform the integration response body; `parameters`: the response parameters from the back-end response that API Gateway sends to the method response. Parameters can be used to set outbound CORS headers: `method.response.header.Access-Control-Allow-Origin: "'*'"` or to map custom dynamic headers: `method.response.header.Custom: "integration.response.body.customValue"`

Example:

```
apigateway:
  helloapi:
    resources:
      /hello:
        method: GET
        integration:
          lambda: helloworld.sayhi
          responses:
```

```
            - pattern: ""
              code: "404"
              template:
                application/json: |
                  #set($inputRoot = $input.path('$'))
                  $inputRoot
```

Full Example

```
apigateway:

    helloapi:

        description: My complex hello API
        resources:
            /:
                methods: GET
                integration:
                    lambda: helloworld.sayhi
            /hi:
                methods: [GET, POST]
                integration:
                    lambda: helloworld.sayhi

            /hi/with-errors:
                method: GET
                integration:
                    lambda: helloworld.sayhi
                    responses:
                        - pattern: ""
                          code: "404"
                responses:
                    - code: "404"

            /hi/none:
                method: GET

            /hi/http:
                methods: GET
                integration:
                    type: HTTP
                    uri: https://www.google.com

            /hi/mock:
                methods: GET
                integration:
                    type: MOCK

            /{integration+}:
                methods: POST
                integration:
                    lambda: helloworld.sayho
                    type: AWS_PROXY

            /parameters:
                methods: GET
```

```
                parameters:
                    method.request.header.color: True
                integration:
                    lambda: helloworld.hellopy
                    responses:
                        - pattern: ""
                          code: "200"
                    parameters:
                        integration.request.querystring.color: method.request.header.
↪color
                responses:
                    - code: "200"
                      parameters:
                          method.response.header.color: color

            /cors:
                methods: GET
                integration:
                    lambda: helloworld.hellopy
                    responses:
                        - pattern: ""
                          code: "200"
                          parameters:
                              method.response.header.Access-Control-Allow-Origin: "'*'
↪"
                              method.response.header.Access-Control-Allow-Methods: "'*
↪'"
                              method.response.header.Access-Control-Request-Method: "
↪'GET'"
                responses:
                    - code: "200"
                      parameters:
                          method.response.header.Access-Control-Allow-Origin: true
                          method.response.header.Access-Control-Allow-Methods: true
                          method.response.header.Access-Control-Request-Method: true

            /hi/complex/:
                methods:
                    GET:
                        integration:
                            lambda: helloworld.sayhi
                    POST:
                        integration:
                            lambda: helloworld.sayhi

            /content-types:
                methods: POST
                integration:
                    lambda: helloworld.sayhi
                    responses:
                        - pattern: ""
                          code: "200"
                          template:
                              application/json: |
                                #set($inputRoot = $input.path('$'))
                                $inputRoot

                request_templates:
```

```
                application/x-www-form-urlencoded: |
                    #set($inputRoot = $input.path('$'))
                    {}
            responses:
                - code: "200"
                  models:
                      application/xml: Empty
```

2.4.2 Dynamodb

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability.

Gordon allow you to integrate your lambdas with dynamodb using their streams service. DynamoDB Streams captures a time-ordered sequence of item-level modifications in any DynamoDB table, and stores this information in a log for up to 24 hours. Applications can access this log and view the data items as they appeared before and after they were modified, in near real time.

Every time one of our dynamodb tables get's modified, dynamo notifies the stream, and one of our lambdas is executed.

Note: As always, is not gordon's business to create the source `stream`. You should create them in advance. You can read Why in the *FAQ*

Anatomy of the integration

```
dynamodb:

  { INTEGRATION_NAME }:
    lambda: { LAMBDA_NAME }
    stream: { ARN }
    batch_size: { INT }
    starting_position: { STARTING_POSITION }
```

Properties

Integration Name

Name	Key of the `dynamodb` map.
Required	Yes
Valid types	`string`
Max length	30
Description	Name for your dynamodb integration. Try to keep it as short and descriptive as possible.

Lambda

Name	`lambda`
Required	Yes
Valid types	`lambda-name`
Description	Name of the lambda you want to notify

Stream

Name	`stream`
Required	Yes
Valid types	`arn`
Description	Arn of the dynamodb stream you want to connect your lambda with.

Batch size

Name	`stream`
Required	No
Default	100
Valid types	`integer`
Min	1
Max	10000
Description	Number of events you want your lambda to receive at once

Starting position

Name	`starting_position`
Required	Yes
Valid Values	`TRIM_HORIZON`, `LATEST`
Description	Number of events you want your lambda to receive at once

- TRIM_HORIZON: Start reading at the last (untrimmed) stream record, which is the oldest record in the shard. In DynamoDB Streams, there is a 24 hour limit on data retention. Stream records whose age exceeds this limit are subject to removal (trimming) from the stream.
- LATEST: Start reading just after the most recent stream record in the shard, so that you always read the most recent data in the shard.

Full Example

```
dynamodb:

  my_dynamodb_integration:

    lambda: app.dynamoconsumer
    stream: arn:aws:dynamodb:eu-west-1:123456789:table/dynamodbexample/stream/2015-11-
↪14T11:18:58.642
    batch_size: 100
    starting_position: LATEST
```

2.4.3 Events

Amazon CloudWatch Events delivers a near real-time stream of system events that describe changes in Amazon Web Services (AWS) resources to AWS Lambda functions as well as trigger events on a pre-determined schedule .

Using simple rules that you can quickly set up, you can match events and route them to one or more target functions.

A full list of available events can be found here: http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/EventTypes.html

Anatomy of the integration

```
events:

  { INTEGRATION_NAME }:
    state: { STATE }
    description: { STRING }
    schedule_expression: { RATE }
    event_pattern: { MAP }
    targets:
      { TARGET_ID }:
        lambda: { LAMBDA_NAME }
        input: { INPUT }
        input_path: { INPUT_PATH }
```

Note: You need to specify either `schedule_expression`, `event_pattern` or both.

Properties

Integration Name

Name	Key of the `events` map.
Required	Yes
Valid types	`string`
Max length	30
Description	Name for your CLoudWatch integration.

State

Name	`state`
Required	No
Valid Values	`ENABLED`, `DISABLED`
Description	Enables or disables this integration

Schedule Expression

Name	`schedule_expression`
Required	No
Valid Types	`rate`
Description	Rate at which your lambda will be scheduled

All scheduled events use UTC time zone and the minimum precision for schedules is 1 minute. CloudWatch Events rates supports the following formats:

- `cron(<Fields>)`
- `rate(<Value> <Unit>)`

For more information about `cron` and `rate`: http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/ScheduledEvents.html

Examples:

- `cron(0 10 * * ? *)` Run at 10:00 am (UTC) every day.
- `cron(0 18 ? * MON-FRI *)` Run at 06:00 pm (UTC) every Monday through Friday.
- `cron(0/15 * * * ? *)` Run every 15 minutes.
- `rate(5 minutes)` Every 5 minutes.
- `rate(1 hour)` Every 1 hour.
- `rate(2 day)` Every 2 day.

Event pattern

Name	`event_pattern`
Required	No
Valid Types	`map`
Description	Pattern structure which matches certain CloudWatch events you are interested in.

Rules use event patterns to select events and route them to targets. A pattern either matches an event or it doesn't. Event patterns are represented as objects with a structure that is similar to that of events.

```
event_pattern:
    source:
        - aws.ec2
    detail:
        state:
            - pending
```

For more information about Events: http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/CloudWatchEventsandEventPatterns.html

Targets

Name	`targets`
Required	No
Valid Types	`map`
Description	Map of target lambdas to connect this event to, as well as optional input and input_path information.

```
targets:
  say_hello:
    lambda: helloworld.hellopy

say_hello:
  lambda: helloworld.hellopy
  input: xxx
  input_path: yyy
```

Full Example

```
events:
  every_night:
    schedule_expression: cron(0 0 * * ? *)
    description: Call example_lambda every midnight.
    state: ENABLED

    targets:
      say_hello:
        lambda: helloworld.hellopy  # Example lambda

  new_asg_instance:
    description: Do something when an autoscaling-group instance-launch happens
    state: ENABLED

    targets:
      say_hello:
        lambda: helloworld.hellopy  # Example lambda

    event_pattern:
        source:
            - aws.autoscaling
        detail:
            LifecycleTransition:
              - autoscaling:EC2_INSTANCE_LAUNCHING
```

2.4.4 Kinesis

Amazon Kinesis Streams enables you to build custom applications that process or analyze streaming data for specialized needs. Amazon Kinesis Streams can continuously capture and store terabytes of data per hour from hundreds of thousands of sources such as website clickstreams, financial transactions, social media feeds, IT logs, and location-tracking events. Kinesis Streams captures a time-ordered sequence of events, and stores this information in a log for up to 7 days.

Gordon allow you to integrate your lambdas with kinesis using their streams service. Every time on event gets published into the kinesis stream, one of our lambdas is executed.

Note: As always, is not gordon's business to create the source `stream`. You should create them in advance. You can read Why in the *FAQ*

Anatomy of the integration

```
kinesis:

  { INTEGRATION_NAME }:
    lambda: { LAMBDA_NAME }
    stream: { ARN }
    batch_size: { INT }
    starting_position: { STARTING_POSITION }
```

Properties

Integration Name

Name	Key of the `kinesis` map.
Required	Yes
Valid types	`string`
Max length	30
Description	Name for your kinesis integration. Try to keep it as short and descriptive as possible.

Lambda

Name	`lambda`
Required	Yes
Valid types	`lambda-name`
Description	Name of the lambda you want to notify

Stream

Name	`stream`
Required	Yes
Valid types	`arn`
Description	Arn of the kinesis stream you want to connect your lambda with.

Batch size

Name	`batch_size`
Required	No
Default	100
Valid types	`integer`
Min	1
Max	10000
Description	Number of events you want your lambda to receive at once

Starting position

Name	`starting_position`
Required	Yes
Valid Values	`TRIM_HORIZON`, `LATEST`
Description	Number of events you want your lambda to receive at once

- TRIM_HORIZON: Start reading at the last (untrimmed) stream record, which is the oldest record in the shard.
- LATEST: Start reading just after the most recent stream record in the shard, so that you always read the most recent data in the shard.

Full Example

```
kinesis:

  my_kinesis_integration:

    lambda: app.kinesisconsumer
    stream: arn:aws:kinesis:eu-west-1:123456789:stream/kinesisexample
    batch_size: 100
    starting_position: LATEST
```

2.4.5 S3

Amazon Simple Storage Service (Amazon S3), provides developers with secure, durable, highly-scalable object storage. Amazon S3 is easy to use, with a simple web service interface to store and retrieve any amount of data from anywhere on the web.

Gordon allow you to integrate your lambdas with S3 using their notification service. The idea is simple, every time an object get's created/deleted, S3, will trigger a notification which you can route to three different services:

- **Lambda**: Your code will be triggered each time something happens. *(1 S3 event = 1 lambda executed)*
- **SQS**: S3 will create a message in a SQS queue every time something happens. Right now there is no any api for Lambdas to consume messages from a sqs queue, but because you need to define all bucket notification in the same place, we need to support this. In the future AWS will might support consuming SQS messages using lambda. *(1 S3 event = 1 message in a queue)*
- **SNS**: S3 will send a message to a SNS Topic. You can subscribe as many lambdas as you want to this topic and process those messages individually *(1 S3 event = N lambdas executed)*

Note: As always, is not gordon's business to create those `sqs` or `sns` resources. You should create them in advance. You can read Why in the *FAQ*

Limitations

This integration, has some limitations because how the AWS API is designed:

- You must define all notifications for a bucket within the same integration.
- Gordon will refuse to configure notifications in bucket if it already has some other notifications configured manually (this is a safe measure).

Anatomy of the integration

```
s3:

  { INTEGRATION_NAME }:
    bucket: { BUCKET_NAME }
    notifications:

      { NOTFICATION_ID }:
        lambda: { LAMBDA_NAME }
```

```
        queue: { QUEUE_NAME }
        topic: { TOPIC_NAME }
        events:
          - { EVENT_NAME }
        key_filters:
          prefix: { STRING }
          suffix: { STRING }
```

Properties

Integration Name

Name	Key of the `s3` map.
Required	Yes
Valid types	`string`
Max length	30
Description	Name for your s3 integration. Try to keep it as short and descriptive as possible.

Bucket

Name	`bucket`
Required	Yes
Valid types	`string`, `reference`
Max length	30
Description	Name of the bucket source of the events

Notifications

Name	`notifications`
Required	Yes
Valid types	`list`
Description	List of notifications to configure.

Notification ID

Name	`id`
Required	Yes
Valid types	`string`
Description	Unique identifier for this notification

Notification Lambda

Name	`lambda`
Required	No
Valid types	`lambda-name`, `arn`
Description	Name of the lambda you want to notify

Note: Each notification can only configure one `lambda`, `queue` or `topic`.

You can reference lambdas by name

```
lambda: app.s3consumer
```

Or by their full `arn`:

```
lambda: arn:aws:lambda:eu-west-1:123456789:function:function-name
```

Notification Queue

Name	`queue`
Required	No
Valid types	`queue-name`, `map`
Description	Name of the queue you want to notify

Note: Each notification can only configure one `lambda`, `queue` or `topic`.

You can reference queues by name if they are in the same account than the bucket

```
queue: my-queue-name
```

If your queue is on a different account you can use the dictionary format:

```
queue:
  name: my-queue-name
  acount_id: 123456789
```

Notification Topic

Name	`topic`
Required	No
Valid types	`topic-name`, `map`
Description	Name of the topic you want to notify

You can reference topics by name if they are in the same account than the bucket

```
topic: my-topic-name
```

If your topic is on a different account you can use the dictionary format:

```
topic:
  name: my-topic-name
  acount_id: 123456789
```

Notification Events

Name	events
Required	Yes
Valid types	list
Description	List of events you want to make trigger a notification

The list of available events is the following:

- s3:ObjectCreated:*
- s3:ObjectCreated:Put
- s3:ObjectCreated:Post
- s3:ObjectCreated:Copy
- s3:ObjectCreated:CompleteMultipartUpload
- s3:ObjectRemoved:*
- s3:ObjectRemoved:Delete
- s3:ObjectRemoved:DeleteMarkerCreated
- s3:ReducedRedundancyLostObject

Note: Remember that you can't overlap events between notifications. So, if you for example subscribe a lambda to s3:ObjectCreated:*, you'll not be able to subscribe any other notification to: s3:ObjectCreated:Put, s3:ObjectCreated:Post, etc...

Key Filters

Name	key_filters
Required	No
Valid types	map
Description	Map of filters you want to apply

Filters are optional to all notifications. The current AWS API only allows you to filter events by the key's prefix and suffix. One notification can't define more than one of each (prefix and suffix) and filters in a bucket can't overlap one to each other.‘‘prefix‘‘ and suffix value and can be either a string or a references.

Full Example

```
s3:
  my_s3_integration:
    bucket: my_bucket_name
    notifications:

      lambda_on_create_cat:
        lambda: app.s3consumer
        events:
          - s3:ObjectCreated:*
        key_filters:
          prefix: cat_
          suffix: .png
```

```
    queue_on_remove_dog:
      queue: removed_dogs_queue
      events:
        - s3:ObjectRemoved:*
      key_filters:
        prefix: dog_

    topic_on_redundacy_lost:
      topic: redundacy_lost_topic
      events:
        - s3:ReducedRedundancyLostObject:*
```

2.5 Parameters

Parameters are the artifact around the fact that project templates (what gordon generates into `_build`) should be immutable between stages. Parameters allow you to have specific settings based on the stage where you are applying your project.

2.5.1 How can I use parameters?

In order to use parameters you only need to:

- Create a directory called `parameters`
- Inside of this directory, create `.yml` files for your different stages (`dev.yml`, `prod.yml`, ...)
- Replace values in your settings with `ref://MyParameter`
- Add values for `MyParameter` in `dev.yml`, `prod.yml` ...

```
...
parameters
- dev.yml
- prod.yml
- common.yml
```

Note: You can create a file called `common.yml`, and place all shared parameters between stages on it. When this file is present, gordon will read it first, and then update the parameters map using your stage-specific settings file (if pressent).

If you want to customize your parameters further, read *Advanced Parameters*, where you'll find information on how make paramater values be dynamic.

2.5.2 Example

Let's imagine you want to call one lambda every time a file is created in one of your buckets.

- First, you'll create and register a lambda.
- Then you'll create a new event source.

Something like this:

```
s3:
  my_s3_integration:
    bucket: my-dev-bucket
    notifications:

      - id: lambda_on_create_cat
        lambda: app.mys3consumer
        events:
          - s3:ObjectCreated:*
```

This is good to start with, but what about when you want to put this on production? You'll need to:

- Change the bucket to `my-production-bucket` instead of `my-dev-bucket`
- Build your project `gordon build`
- Apply the project into production `gordon apply --stage=prod`

But... now every time you want to develop the lambda further in your `dev` stage, you'll need to change the bucket... again and again back and forth.

This is tedious and unmaintainable.

2.5.3 Solution

The solution is as simple as making your bucket name be a parameter. In this case we are calling the parameter `MyS3Bucket`.

```
s3:
  my_s3_integration:
    bucket: ref://MyS3Bucket
    notifications:

      - id: lambda_on_create_cat
        lambda: app.mys3consumer
        events:
          - s3:ObjectCreated:*
```

Then, in the root of you project, create a new directory called `parameters` and create a new file with the name of each of the stages (`dev` and `prod`).

```
...
parameters/
- prod.yml
- dev.yml
```

Then, we can define two different values for `MyS3Bucket` based on the stage where we are applying the project.

`prod.yml` will have the production bucket:

```
---
MyS3Bucket: my-production-bucket
```

and `dev.yml` will have the dev one:

```
---
MyS3Bucket: my-dev-bucket
```

Now we can simply run:

- `gordon apply --stage=dev`
- `gordon apply --stage=prod`

And the correct settings will be used.

2.5.4 How it works?

When you define in your settings file a value as a reference `ref://`, gordon will automatically register (on `build` time) all required input parameters in your CloudFormation templates and collect values from your parameters files when you call `apply`.

Remember that you can create a file called `common.yml`, and place all shared parameters between stages on it.

2.6 Contexts

Contexts in Gordon are groups of variables which you want to make accessible to your code, but you don't want to hardcode into it because it's values are dependant on the the deployment.

This could be for example because `dev` and `production` lambdas (although beeing the same code), need to connect to different resources, use different passwords or produce slightly different outputs.

In the same way, same lambdas deployed to different regions will probably need to connect to different places.

Contexts solve this problem by injecting a small payload into the lambdas package on deploy time, and letting you read that file on run time using your language of choice.

2.6.1 How contexts works

The first thing you'll need to do is define a context in your project settings file (`project/settings.yml`).

```
---
project: my-project
default-region: eu-west-1
code-bucket: my-bucket
apps:
  ...
contexts:
  default:
    database_host: 10.0.0.1
    database_username: dev-bob
    database_password: shrug
...
```

As you can see, we have defined a context called `default`. All lambdas by default inject the context called `default` if it is present.

After doing this, Gordon will leave a `.context` JSON file at the root of your lambda package. You can use your language of choice to read and use it.

In the following example, we use python to read this file.

```
import json

def handler(event, context):
    with open('.context', 'r') as f:
```

```
        gordon_context = json.loads(f.read())
    return gordon_context['database_host']  # Echo the database host
```

Same example, but written in Javascript:

```
var gordon_context = JSON.parse(require('fs').readFileSync('.context', 'utf8'));

exports.handler = function(event, context) {
    context.succeed(gordon_context['database_host']);  // Echo the database host

};
```

And Java:

```
// Remember to add 'org.json:json:20160212' to your gradle file
package example;

import java.io.FileNotFoundException;
import java.util.Scanner;
import java.io.File;
import com.amazonaws.services.lambda.runtime.Context;
import org.json.JSONObject;


public class Hello {

    public static class EventClass {
        public EventClass() {}
    }

    public String handler(EventClass event, Context context) throws␣
↪FileNotFoundException{
        JSONObject gordon_context = new JSONObject(
            new Scanner(new File(".context")).useDelimiter("\\A").next()
        );
        return gordon_context.getString("database_host");
    }

}
```

2.6.2 Advanced contexts

For obvious reasons, hardcoding context values in your `project/settings.yml` file is quite limited and not very flexible. For this reason Gordon allows you to make the value of any of the context variables reference any parameter.

In the following example, we are going to make all three variables point to three respective parameters. This will allow us to change the value of the context variables easily between stages or regions.

```
---
project: my-project
default-region: eu-west-1
code-bucket: my-bucket
apps:
  ...
contexts:
  default:
```

```
    database_host: ref://DatabaseHost
    database_username: ref://DatabaseUsername
    database_password: ref://DatabasePassword
...
```

Now we only need to define what is the value for each of these parameters creating (for example) a `parameters/common.yml` file

```
---
DatabaseHost: 10.0.0.1
DatabaseUsername: "{{ stage }}-bob"
DatabasePassword: env://MY_DATABASE_PASSWORD
```

As you can see this is quite a fancy example, because values are now dynamically generated.

Parameter	Value
`DatabaseHost`	This is a fixed hardcoded value `10.0.0.1`.
`DatabaseUsername`	This is a jinja2 parameter. If we apply our project into a stage called `prod` it's value will be `prod-bob`
`DatabasePassword`	This parameter will have as value whatever the `MY_DATABASE_PASSWORD` env variable has when you apply your project.

Now you should have a basic understanding of how contexts works. If you want to learn more about `parameters` you'll find all the information you need in:

- *Parameters* How parameters works
- *Advanced Parameters* Advanced usages of parameters.

2.7 Running lambdas locally

While developing lambdas it is quite useful to be able to run lambdas locally and see how they behave when receiving certain events. This should not be consider a replacement for writing tests - You should write tests for your code!

In order to locally invoke your lambdas you can do so by running:

```
$ echo '{... JSON ...}' | gordon run APP.LAMBDA
```

Gordon expects `stdin` to be the json formated event your lambda will receive. It is important to note that your lambda will be executed after collecting all resources and applying the full `build` process, so you can expect dependencies to be available.

2.7.1 Python lambdas

Python lambdas don't require any specific setup, but you should keep in mind the limitations of of the mock `LambdaContext` object that gordon uses as second argument of your lambda. You can find the current implementation Python Loader.

We'll try to make this mock more clever overtime. PR Welcome!

2.7.2 Node Lambdas

Node lambdas don't require any specific setup, but you should keep in mind the limitations of of the mock `LambdaContext` object that gordon uses as second argument of your lambda. You can find the current imple-

mentation Node Loader.

We'll try to make this mock more clever overtime. PR Welcome!

2.7.3 Java Lambdas

Java lambdas require you to write an adapter which accepts a `String` as the first argument and `Context` as second.

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import org.json.JSONObject;


public class Hello {

    public static class EventClass {

        ...

        public EventClass(String key1, String key2, String key3) {
            this.key1 = key1;
            this.key2 = key2;
            this.key3 = key3;
        }

    }

    public String handler(EventClass event, Context context) {
        System.out.println("value1 = " + event.key1);
        System.out.println("value2 = " + event.key2);
        System.out.println("value3 = " + event.key3);
        return String.format(event.key1);
    }

    public String handler(String json_event, Context context) {
        JSONObject event_data = new JSONObject(json_event);
        EventClass event = new EventClass(
            event_data.getString("key1"),
            event_data.getString("key2"),
            event_data.getString("key3")
        );
        return this.handler(event, context);
    }

}
```

As you can see we have defined an adapter with the following signature `public String handler(String json_event, Context context)` which calls our lambda handler after creating a `EventClass` instance using the data from the json in `json_event`.

In a similar way than Python and Javascript lambdas you should keep in mind the limitations of of the `MockContext` object that gordon uses as second argument of your lambda. You can find the current implementation Java Loader.

We'll try to make this mock more clever overtime. PR Welcome!

2.8 Running lambdas in AWS

Once you have deployed your lambdas to AWS, it might me interesting for you to run them from your command line. After running `$ gordon apply`, you'll get the full arn of each of the lambdas you deployed.

In order to do so, you can use the official `aws-cli` tool

```
$ aws lambda invoke \
--function-name $ARN \
--log-type Tail \
--payload '{"key1":"value1", "key2":"value2", "key3":"value3"}' \
output.txt \
| jq -r .LogResult | base64 --decode
```

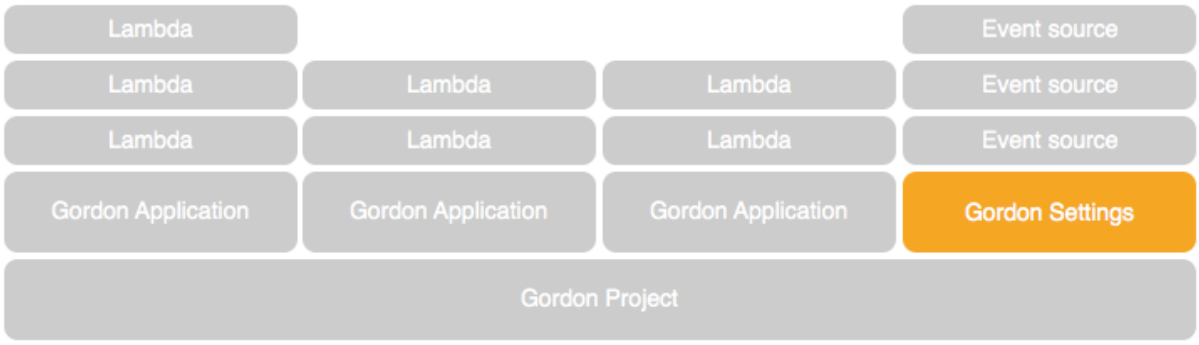
As you can see we use `jq` in order to slice the output JSON and `base64` to decode it.

CHAPTER 3

In detail

In detail explanations and advanced use cases.

3.1 Settings



Gordon `settings.yml` files are simple yaml files which define how Gordon should behave.

Settings can be defined at two different levels, project and application level.

Resources such as lambdas or event sources can be defined in both levels, but there are some other settings which are only expected at project level.

3.1.1 General

These settings can be defined either at project or app level.

Settings	Description
`lambdas`	Lambda definitions. *Anatomy of Lambdas*
`cron`	CloudWatch Events definitions. *Anatomy of Cron integration*
`dynamodb`	Dynamodb Event Source definitions. *Anatomy of Dynamodb integration*
`kinesis`	Kinesis Event Source definitions. *Anatomy of Kinesis integration*
`s3`	S3 notfication definitions. *Anatomy of S3 integration*
`apigateway`	API Gateway definitions. *Anatomy of Api Gateway integration*
`aws-account-id`	AWS account id where you are deploying your lambdas. If not present, gordon will try to retrieve it using IAM api.

3.1.2 Project settings

Project settings are defined in the root level of your project (`project/settings.yml`). The section *Project Anatomy* will give you more information about how you can customize your project.

3.1.3 Application settings

Application settings are defined within your applications (`application/settings.yml`).

One particularity about application settings, is that those can be redefined as part of it's initialization. If for example you want to reuse one application and redefining some settings, you can do so:

```
---
project: my-project
default-region: eu-west-1
code-bucket: my-bucket
apps:
  - exampleapp
  - exampleapp:
      a: b
      c: d
...
```

3.2 Advanced Parameters

It is great that you can abstract your project to use different parameters on each stage, but sometimes the fact that those parameters are static makes quite hard to describe your needs.

There are two ways you can customize your parameters further, Protocols and Templates.

3.2.1 Protocols

Protocols are helpers which will allows you to make the value of one parameter be dynamic. Protocols are evaluated on `apply` time.

Environment Variables

You can make the value of your parameter be based on any environment variable using the `env://` protocol.

```
---
MyParameter: env://MY_ENV_VARIABLE
```

gordon will make the parameter `MyParameter` value be whatever `MY_ENV_VARIABLE` value is on `apply` time.

Dynamodb

You can dynamically lookup for dynamodb tables which name starts with, ends with or matches certain text.

Name	Description
`dynamodb-startswith://`	Table name startswith certain text
`dynamodb-endswith://`	Table name ends with certain text
`dynamodb-match://`	Table name match certain regular expression

Example:

```
---
MyParameter: dynamodb-startswith://clients-
```

gordon will make the parameter `MyParameter` value be the full name of the table which name starts with `clients-`.

If several (or none) dynamodb tables match any of these criterias, gordon will fail before trying to apply this project.

Dynamodb Streams

You can dynamically lookup for dynamodb streams which table name starts with, ends with or matches certain text.

Name	Description
`dynamodb-stream-startswith://`	Table name startswith certain text
`dynamodb-stream-endswith://`	Table name ends with certain text
`dynamodb-stream-match://`	Table name match certain regular expression

Example:

```
---
MyParameter: dynamodb-stream-startswith://clients-
```

gordon will make the parameter `MyParameter` value be the `ARN` of the stream of which table name starts with `clients-`.

If several (or none) dynamodb tables match any of these criterias, gordon will fail before trying to apply this project.

Kinesis

You can dynamically lookup for kinesis streams which name starts with, ends with or matches certain text.

Name	Description
`kinesis-startswith://`	Kinesis stream name startswith certain text
`kinesis-endswith://`	Kinesis stream name ends with certain text
`kinesis-match://`	Kinesis stream name match certain regular expression

Example:

```
---
MyParameter: kinesis-startswith://events-
```

gordon will make the parameter `MyParameter` value be the full name of the table which name starts with `events-`.

If several (or none) kinesis streams match any of these criterias, gordon will fail before trying to apply this project.

3.2.2 Jinja2 Templates

If you want to customize your parameter values even further, you can use jinja2 syntax to customize the value of your parameters.

The context gordon will provide to this jinja helper is:

- `stage`: The name of the stage where you are applying your project.
- `aws_region`: The name of the AWS_REGION where you are applying your project.
- `aws_account_id`: The ID of the account that you are using to apply your project.
- `env`: All available environment variables.

Example:

```
---
MyBucket: "company-{{ stage }}-images"
```

There are lot's of things you can do with Jinja2. For more information Jinja2 Template Designer Documentation

3.3 gordon.contrib

When using gordon, you'll quickly see `contrib` apps take an important role while deploying and wiring your lambdas.

Gordon uses `CloudFormation` a lot. It is a great AWS service, but it's API doesn't always include the latest services AWS is releasing almost every week! These services will eventually make it into `CloudFormation`, but in the meantime we need to use low-level APIs to interact with them.

We could (as some other projects) decide to fill the gaps streaming API commands... but we decided to do things differently.

We believe `CloudFormation` is the way to move forward, and the advantages it provides surpass some of the gotchas, so we decided to fill the gaps using Lambdas and custom `CloudFormation` resources.

But... how could you use Lambdas and `CloudFormation` to create Lambdas in `CloudFormation`?

Easy - eating your own dog food; That's `gordon.contrib`!

`gordon.contrib` is a set of reusable gordon applications your gordon projects use to be able to deploy your resources and fill the gaps in `CloudFormation` until AWS fills them.

3.3.1 When will AWS allow us to create those resources "natively"?

We have no idea, but we have make a big effort trying to make our Custom `CloudFormation` resources look as similar as possible to what we think those resources will look like. Once AWS releases those APIs in `CloudFormation`, subsequent versions of gordon will stop using our lambda-based Resources and use native ones.

3.3.2 Available `contrib` apps

`contrib.lambdas`

This application exposes two `CloudFormation` resources:

- **version**: Creates Versions for our lambda functions. Nothing super fancy under the hood: https://github.com/jorgebastida/gordon/blob/master/gordon/contrib/lambdas/version/version.py

`contrib.s3`

This application exposes one `CloudFormation` resource:

- **bucket_notification_configuration**: This resource allows us to manage S3 bucket notifications. This is a complex resource because the API AWS has develop around... it is not very nice (imho). You can see more details here: https://github.com/jorgebastida/gordon/blob/master/gordon/contrib/s3/bucket_notification_configuration/bucket_notification_configuration.py

`contrib.helpers`

This application exposes one simple `CloudFormation` resource called `Sleep`. Yes, this is lame `¯\_()_/¯` but it was the only possible way to make resilient integrations with streams such as `kinesis` and `dynamodb`.

This is because the IAM role of the lambda is not propagated fast enough uppon creation, and `CloudFormation` checks if the referenced lambda has permission to consume this stream on creation time. A small sleep fixes the problem. We will probably try fix this in the future with a generic `make-sure-this-is-ready` lambda.

3.4 Setup AWS Credentials

There are few ways and things to consider in order to configure your AWS credentials in your machine.

- http://docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-started.html
- http://docs.aws.amazon.com/general/latest/gr/aws-access-keys-best-practices.html
- https://boto3.readthedocs.io/en/latest/guide/configuration.html

3.5 FAQ

3.5.1 Why my project has some lambdas I've not defined?

You'll get an in-depth explanation of what these lambdas are in the *gordon.contrib* section, but the tl;dr version is that gordon needs those in order to be able to only use CloudFormation to create your resources.

3.5.2 Why gordon don't let me create other resources

Because it would be a terrible idea.

3.5.3 How can I read the logs generated by my Lambdas?

There are two options:

- You can read them online in your CloudWatch Logs Console.
- You can use awslogs from your command line.

CHAPTER 4

Tutorials

If you like step by step tutorials... this is your place!

4.1 My first Javascript Lambda

In this example we are going to create our first javascript lambda using gordon. This lambda is going to do the same than the `Hello World` example AWS provides as blueprint. This is:

- Receive an input message
- Log `key1`, `key2` and `key3` values.
- Return `key1` as result.

The test message we'll use to test this function is the following:

```
{
  "key3": "value3",
  "key2": "value2",
  "key1": "value1"
}
```

Before we start, make sure you have:

- An AWS Account
- You've setup your AWS credentials in your machine (*Setup AWS Credentials*)
- Gordon is installed (*Installation*)

4.1.1 Create your Project

From the command line, cd into a directory where you'd like to store your code, then run the following command:

```
$ gordon startproject hellojs
```

This will create a *hellojs* directory in your current directory with the following structure:

```
hellojs
- settings.yml
```

This is the minimal layout of a project. We are now going to create an application.

4.1.2 Create your Application

Run the following command from the command line:

```
$ gordon startapp firstapp --runtime=js
```

This will create a *firstapp* directory inside your project with the following structure:

```
firstapp/
- helloworld.js
- settings.yml
```

By default, when you create a new application, gordon will create one really simple lambda called `helloworld`.

In the next step we'll install your application

4.1.3 Install your application

In order to install your application you need to add it to the `apps` list in the project `settings.yml`.

Edit your project `settings.yml` file and add `firstapp` to the list of installed apps.

```
---
project: hellojs
default-region: us-east-1
apps:
  - gordon.contrib.helpers
  - gordon.contrib.lambdas
  - firstapp
```

In the next step we are going to make the default lambda gordon provides, do what we want it to do.

4.1.4 Create your Lambda

Open you `firstapp/helloworld.js` file and edit it until it looks to something like this:

```
exports.handler = function(event, context) {
  console.log('value1 =', event.key1);
  console.log('value2 =', event.key2);
  console.log('value3 =', event.key3);
  context.succeed(event.key1);  // Echo back the first key value
};
```

The code of our lambda is ready! We only need to double check it is correctly registered.

Open your `firstapp/settings.yml`. It should look similar to this:

```
lambdas:
  helloworld:
    code: helloworld.js
    #description: Simple functions in js which says hello
    #handler: handler
    #role:
    #memory:
```

This file is simply registering a lambda called `helloworld`, and telling gordon the source of the lambda is in `helloworld.js` file.

The default behaviour for gordon is to assume the function to call in your source file is called `handler`. You can change this behaviour by changing the `handler` section in your lambda settings.

Now we are ready to build your project!

4.1.5 Build your project

In the root of your project run the following command

```
$ gordon build
```

This command will have an output similar to:

```
$ gordon build
Loading project resources
Loading installed applications
  contrib_helpers:
    ✓ sleep
  contrib_lambdas:
    ✓ alias
    ✓ version
  firstapp:
    ✓ helloworld
Building project...
  ✓ 0001_p.json
  ✓ 0002_pr_r.json
  ✓ 0003_r.json
```

If that's the case... great! Your project is ready to be deployed.

4.1.6 Deploy your project

Projects are deployed by calling the command `apply`. Apply will assume by default you want to deploy your project into a new stage called `dev`.

Stages are 100% isolated deployments of the same project. The idea is that the same project can be deployed in the same AWS account in different stages (`dev`, `staging`, `production`...) in order to SAFELY test your lambda behaviour.

If you don't provive any stage using `--stage=STAGE_NAME` a default stage called `dev` will be used.

Once you are ready, call the following command:

```
$ gordon apply
```

This command will have an output similar to:

```
$ gordon apply
Applying project...
  0001_p.json (cloudformation)
    CREATE_COMPLETE waiting... -
  0002_pr_r.json (custom)
    ✓ code/contrib_helpers_sleep.zip (364c5f6d)
    ✓ code/contrib_lambdas_alias.zip (e906090e)
    ✓ code/contrib_lambdas_version.zip (c3137e97)
    ✓ code/firstapp_helloworld.zip (db6f502e)
  0003_r.json (cloudformation)
    CREATE_COMPLETE
```

And you are done! Your lambda is ready to be used on AWS!

4.1.7 Test your Lambda

In order to test it, you can navigate into your Lambda Console and:

- Click on the lambda we have just created. It should be called something like: `dev-hellojs-r-FirstappHelloworld-XXXXXXXX`
- Click the blue button named `Test`
- Select the `Hello World` Sample event template (It should come selected by default)
- Click `Save and Test`
- You should get a succeed message: `Execution result: succeeded`, and some log information.

Congratulations! You've just deployed your first lambda into AWS using gordon!

4.2 My first Python Lambda

In this example we are going to create our first python lambda using gordon. This lambda is going to do the same than the `Hello World` example AWS provides as blueprint. This is:

- Receive an input message
- Log `key1`, `key2` and `key3` values.
- Return `key1` as result.

The test message we'll use to test this function is the following:

```
{
  "key3": "value3",
  "key2": "value2",
  "key1": "value1"
}
```

Before we start, make sure you have:

- An AWS Account
- You've setup your AWS credentials in your machine (*Setup AWS Credentials*)
- Gordon is installed (*Installation*)

4.2.1 Create your Project

From the command line, cd into a directory where you'd like to store your code, then run the following command:

```
$ gordon startproject hellopython
```

This will create a *hellopython* directory in your current directory with the following structure:

```
hellopython
- settings.yml
```

This is the minimal layout of a project. We are now going to create an application.

4.2.2 Create your Application

Run the following command from the command line:

```
$ gordon startapp firstapp
```

This will create a *firstapp* directory inside your project with the following structure:

```
firstapp/
- helloworld.py
- settings.yml
```

By default, when you create a new application, gordon will create one really simple lambda called `helloworld`.

In the next step we'll install your application

4.2.3 Install your application

In order to install your application you need to add it to the `apps` list in the project `settings.yml`.

Edit your project `settings.yml` file and add `firstapp` to the list of installed apps.

```
---
project: hellopython
default-region: us-east-1
apps:
  - gordon.contrib.helpers
  - gordon.contrib.lambdas
  - firstapp
```

In the next step we are going to make the default lambda gordon provides, do what we want it to do.

4.2.4 Create your Lambda

Open you `firstapp/helloworld.py` file and edit it until it looks to something like this:

```
from __future__ import print_function
import json

def handler(event, context):
    print("value1 = " + event['key1'])
    print("value2 = " + event['key2'])
```

```
    print("value3 = " + event['key3'])
    return event['key1']  # Echo back the first key value
```

The code of our lambda is ready! We only need to double check it is correctly registered.

Open your `firstapp/settings.yml`. It should look similar to this:

```
lambdas:
  helloworld:
    code: helloworld.py
    #description: Simple functions in python which says hello
    #handler: handler
    #role:
    #memory:
```

This file is simply registering a lambda called `helloworld`, and telling gordon the source of the lambda is in `helloworld.py` file.

The default behaviour for gordon is to assume the function to call in your source file is called `handler`. You can change this behaviour by changing the `handler` section in your lambda settings.

Now we are ready to build your project!

4.2.5 Build your project

In the root of your project run the following command

```
$ gordon build
```

This command will have an output similar to:

```
$ gordon build
Loading project resources
Loading installed applications
  contrib_helpers:
    ✓ sleep
  contrib_lambdas:
    ✓ alias
    ✓ version
  firstapp:
    ✓ helloworld
Building project...
  ✓ 0001_p.json
  ✓ 0002_pr_r.json
  ✓ 0003_r.json
```

If that's the case... great! Your project is ready to be deployed.

4.2.6 Deploy your project

Projects are deployed by calling the command `apply`. Apply will assume by default you want to deploy your project into a new stage called `dev`.

Stages are 100% isolated deployments of the same project. The idea is that the same project can be deployed in the same AWS account in different stages (`dev`, `staging`, `production`...) in order to SAFELY test your lambda behaviour.

If you don't provive any stage using `--stage=STAGE_NAME` a default stage called `dev` will be used.

Once you are ready, call the following command:

```
$ gordon apply
```

This command will have an output similar to:

```
$ gordon apply
Applying project...
  0001_p.json (cloudformation)
    CREATE_COMPLETE waiting... -
  0002_pr_r.json (custom)
    ✓ code/contrib_helpers_sleep.zip (364c5f6d)
    ✓ code/contrib_lambdas_alias.zip (e906090e)
    ✓ code/contrib_lambdas_version.zip (c3137e97)
    ✓ code/firstapp_helloworld.zip (db6f502e)
  0003_r.json (cloudformation)
    CREATE_COMPLETE
```

And you are done! Your lambda is ready to be used on AWS!

4.2.7 Test your Lambda

In order to test it, you can navigate into your Lambda Console and:

- Click on the lambda we have just created. It should be called something like: `dev-hellopython-r-FirstappHelloworld-XXXXXXXX`
- Click the blue button named `Test`
- Select the `Hello World` Sample event template (It should come selected by default)
- Click `Save and Test`
- You should get a succeed message: `Execution result: succeeded`, and some log information.

Congratulations! You've just deployed your first lambda into AWS using gordon!