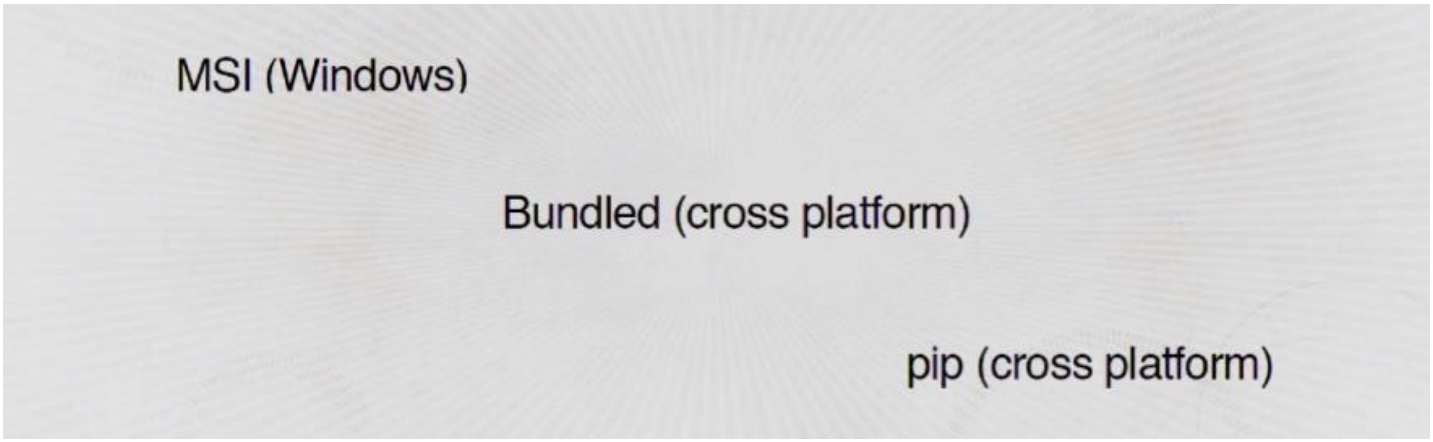
The AWS CLI provides an easy-to-use command line interface to AWS and allows you to create powerful automation scripts. In this session, you learn advanced techniques that open up new scenarios for using the CLI. We demonstrate how to filter and transform service responses, how to chain and script commands, and how to write custom plugins.



There are 3 installation methods for the AWS CLI, it is written in Python with ***pip install awscli***

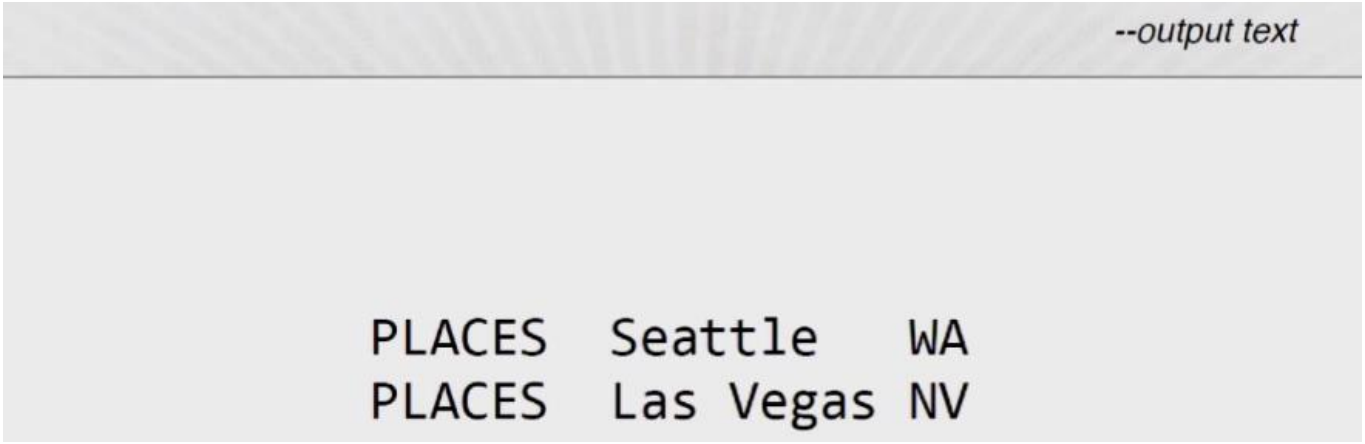
Every command has a similar structure as above.

--output json

```
{
    "Places": [
        {
            "City": "Seattle",
            "State": "WA"
        },
        {
            "City": "Las Vegas",
            "State": "NV"
        }
    ]
}
```

By default, when you run a command you will get JSON output as above.

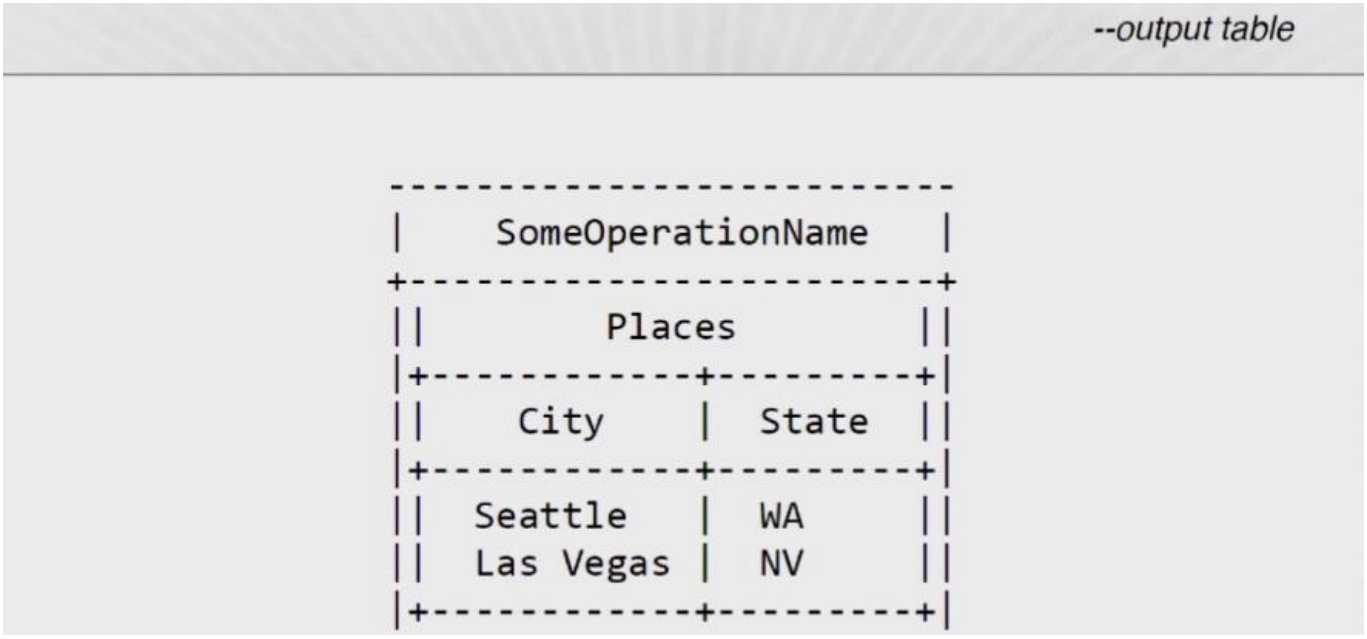
--output text



```
PLACES  Seattle   WA
PLACES  Las Vegas NV
```

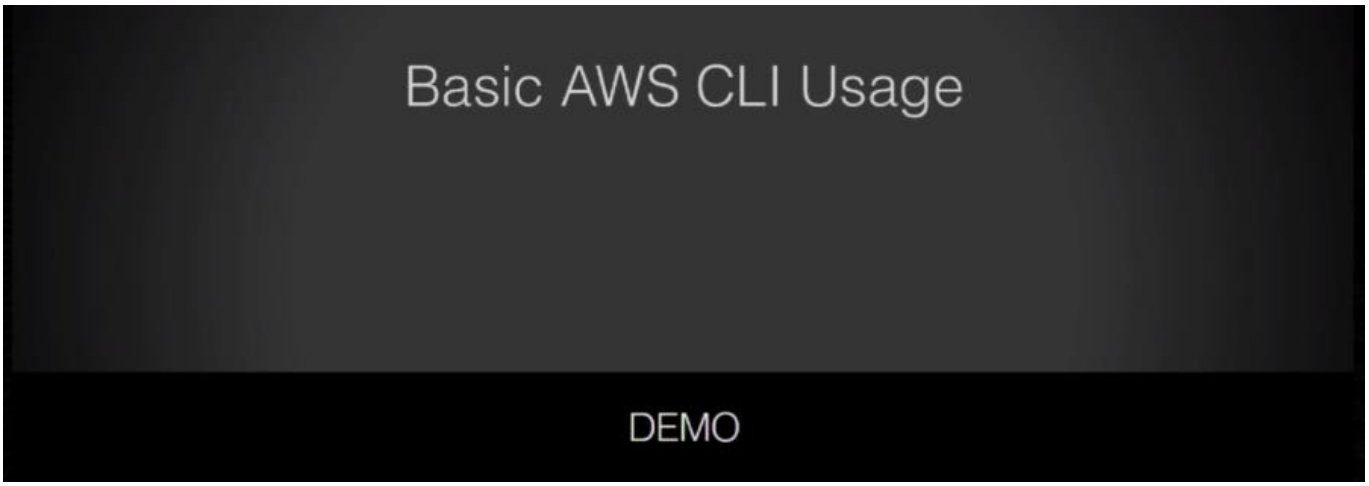
There is also a text output format with results that are tab separated columns as above

--output table

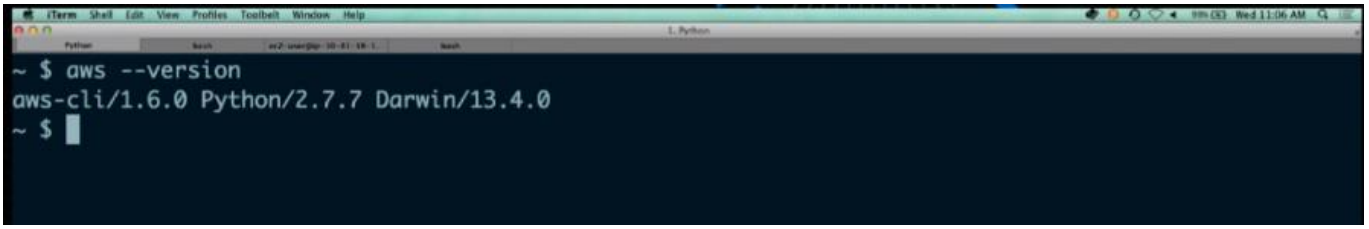


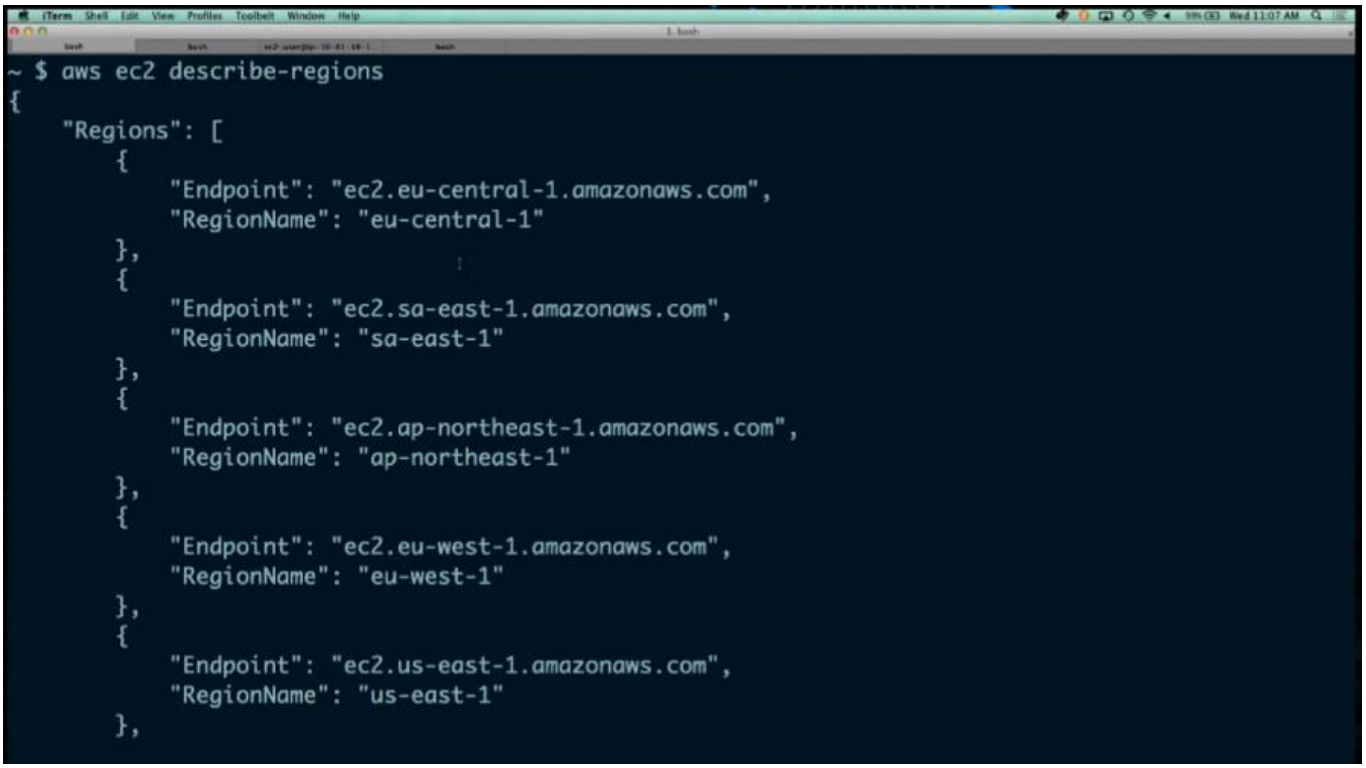
```
-------------------------
|    SomeOperationName   |
+------------------------+
||        Places        ||
|+------------+---------+|
||    City    |  State  ||
|+------------+---------+|
||  Seattle   |  WA     ||
||  Las Vegas |  NV     ||
|+------------+---------+|
```

You also have a table output as above. Table output is good for letting you parse the JSON output visually

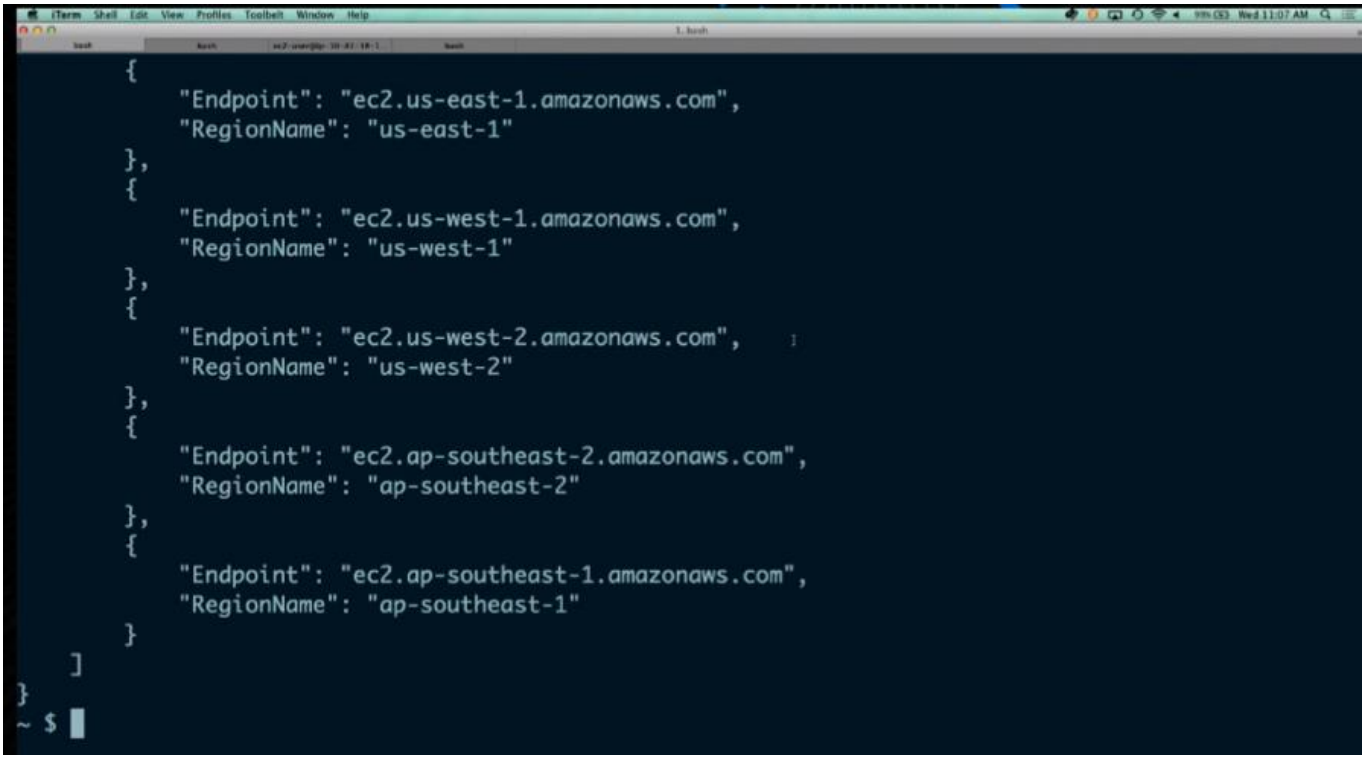


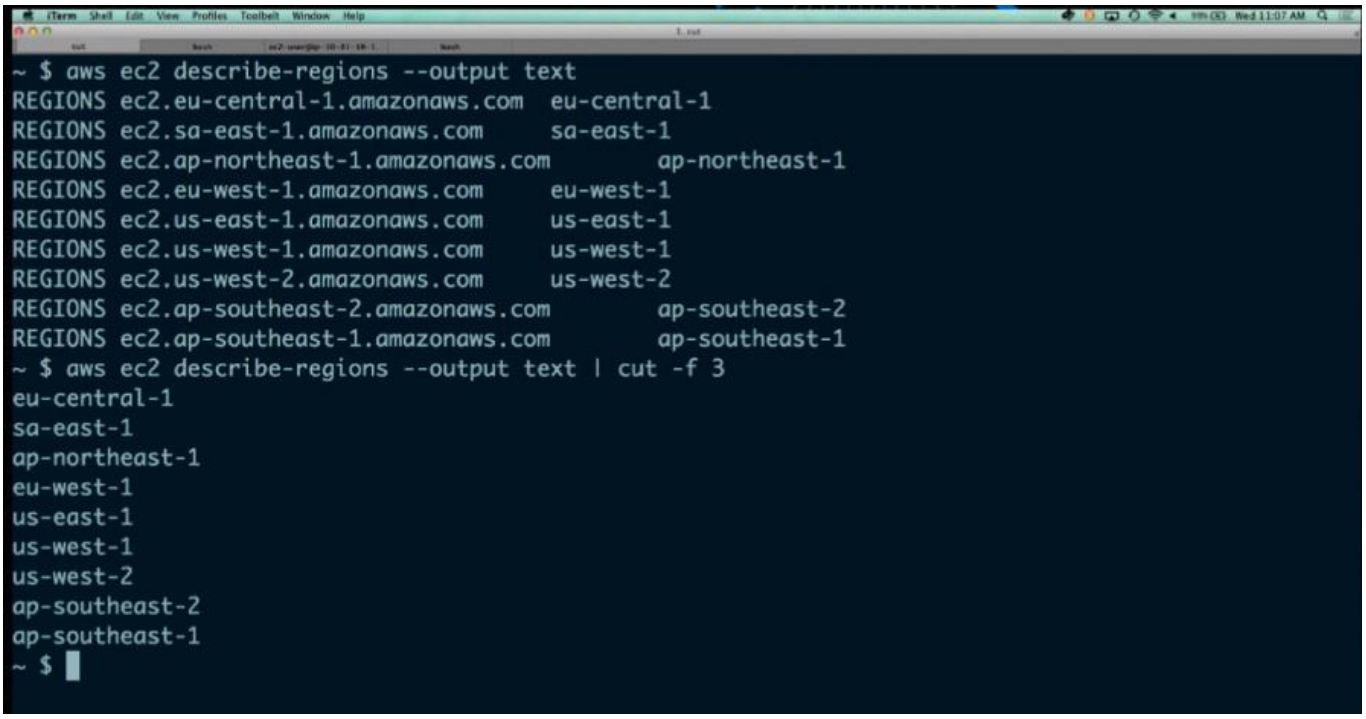
This is how we install ***$ pip install awscli*** and configure ***$ aws configure*** the AWS CLI on our machine for use.



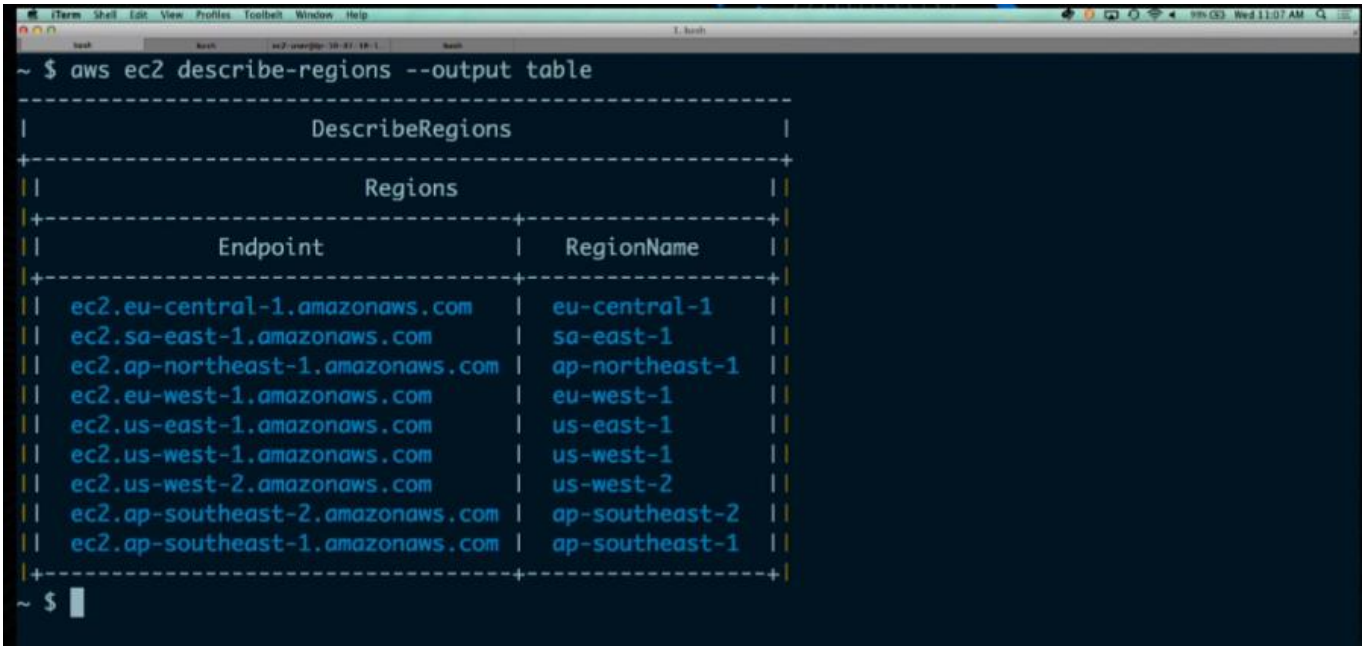


We can use the command ***$ aws ec2 describe-regions***

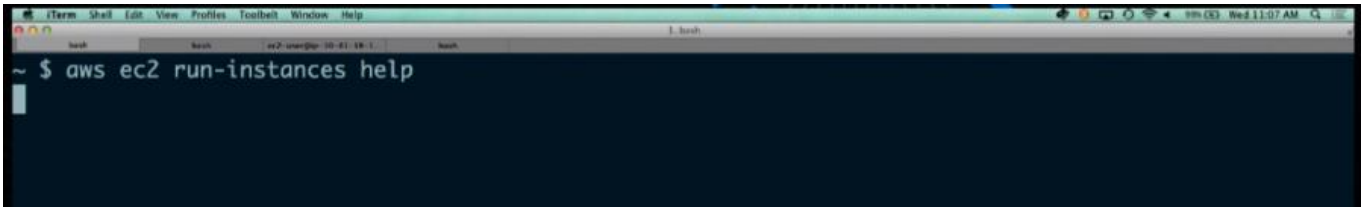




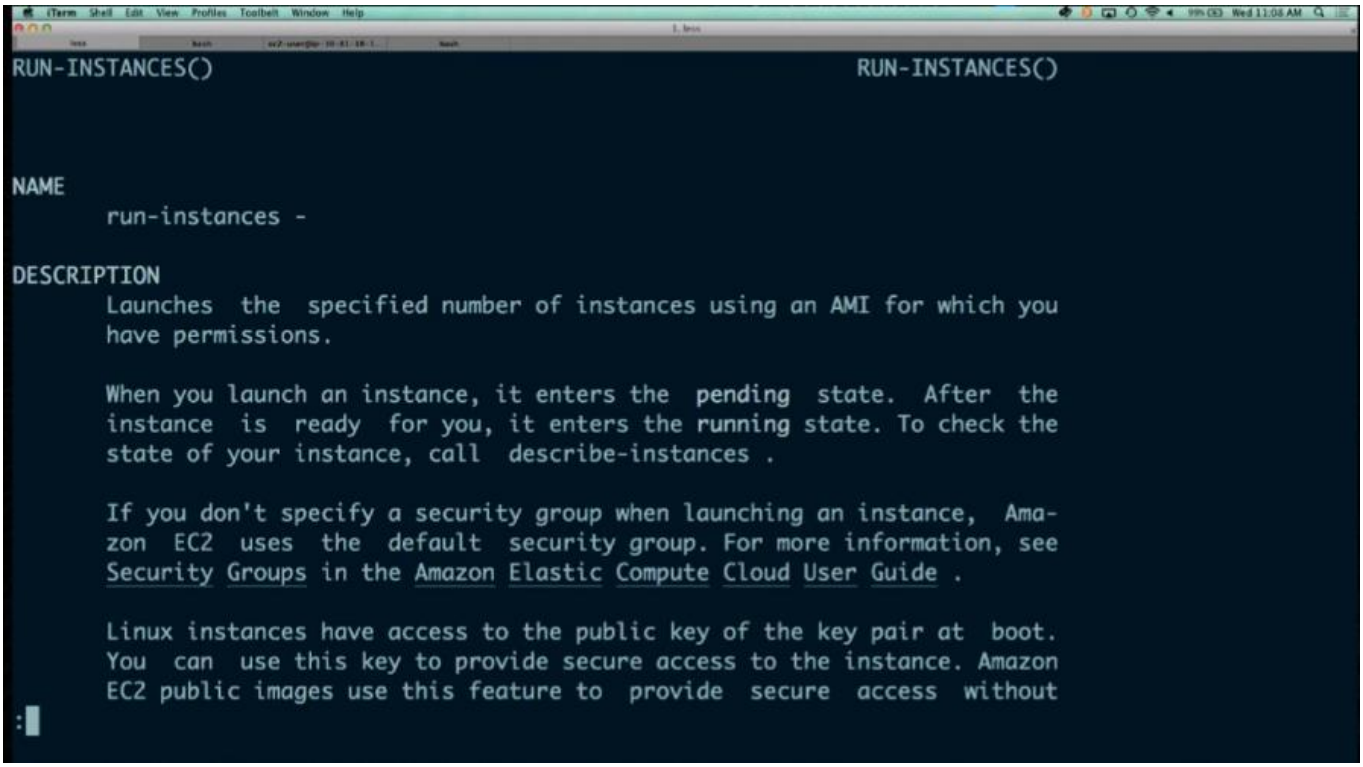
We can use the command for text output with ***$ aws ec2 describe-regions - -output text*** and then get the third column with ***$ aws ec2 describe-regions - -output text | cut -f 3*** as above

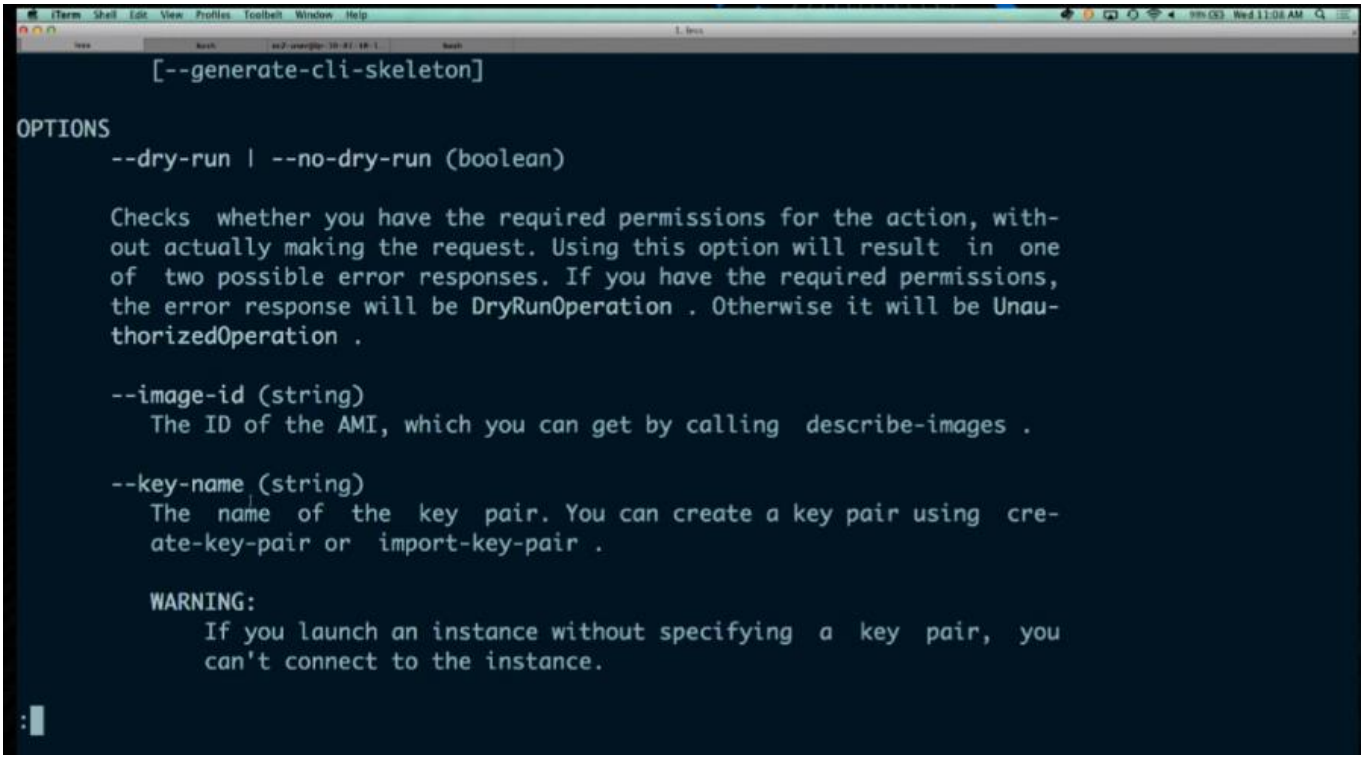


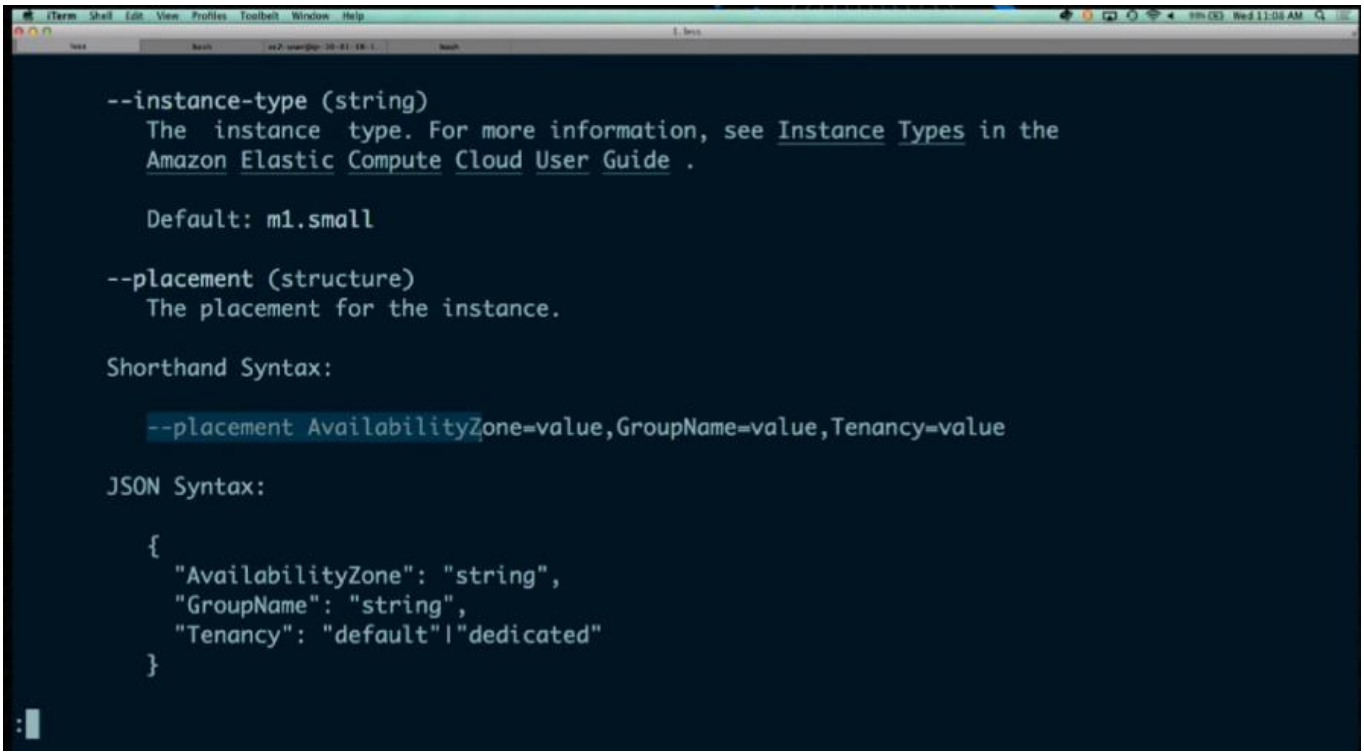
The table output option ***$ aws ec2 describe-regions - -output table*** is visually easy to parse as above

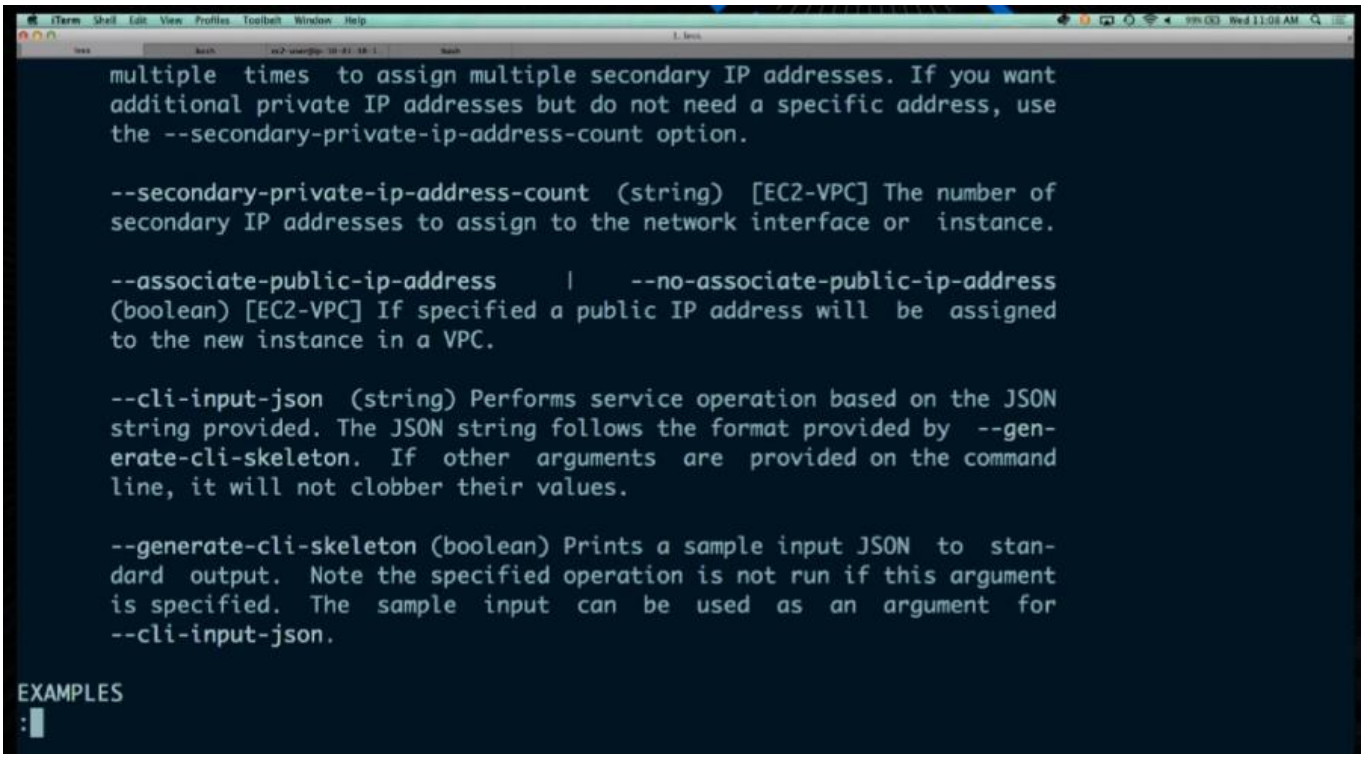


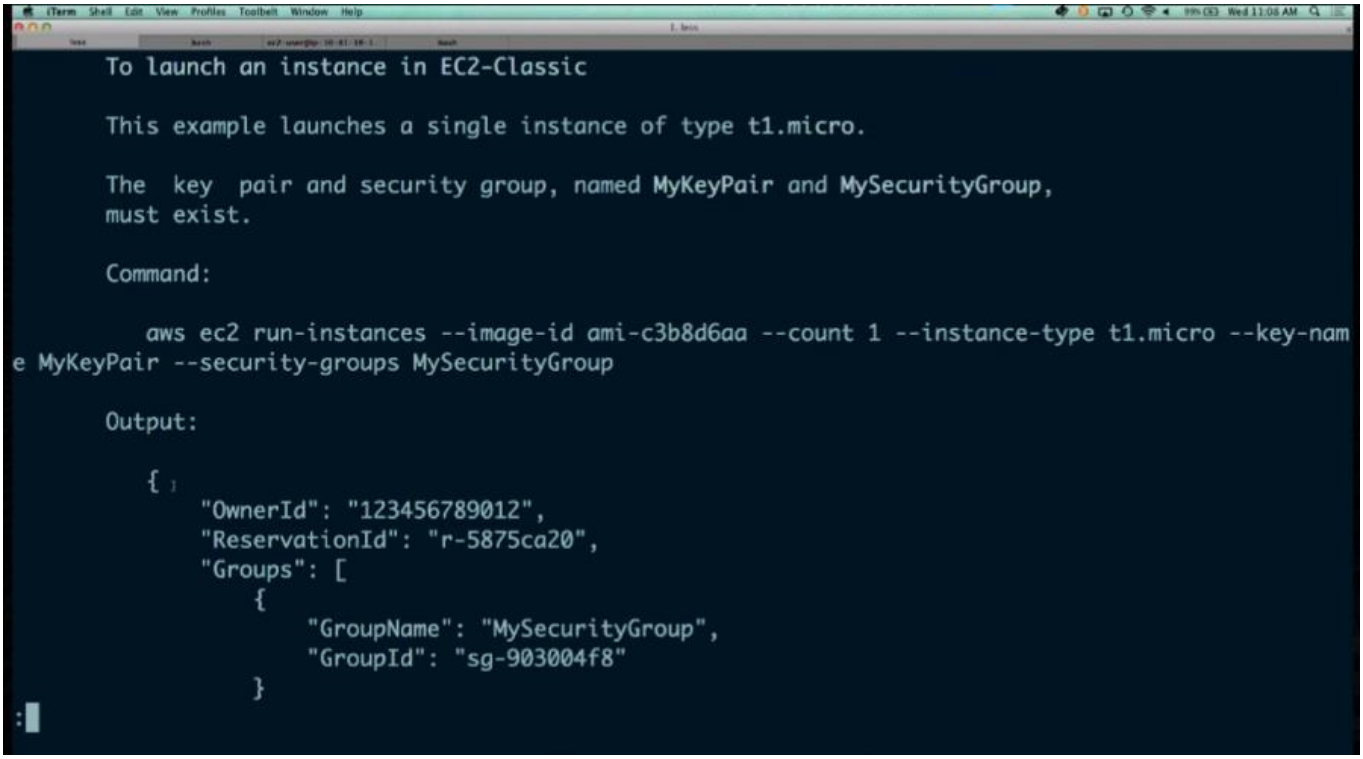
For any command, you can use the help command with it to see the man page as below

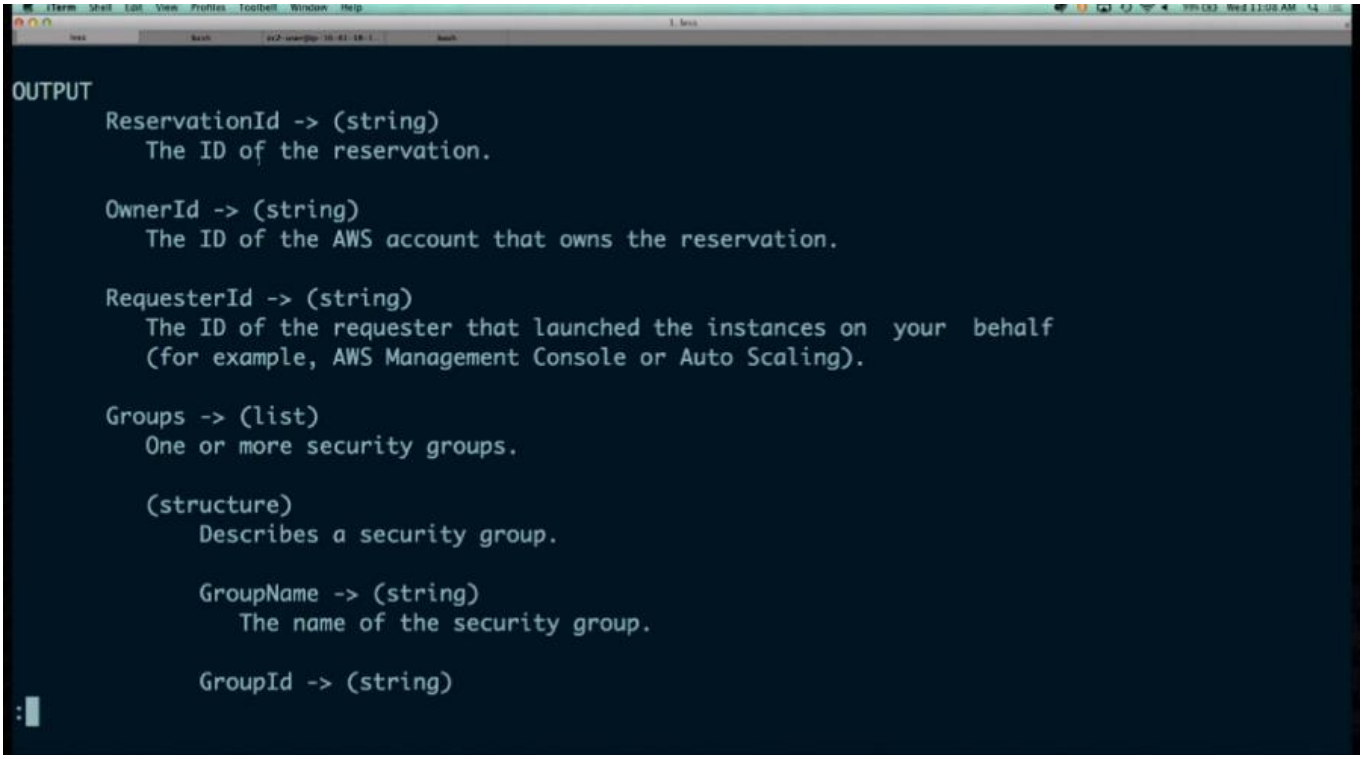

RUN-INSTANCES()
RUN-INSTANCES()
NAME
run-instances -
DESCRIPTION
Launches the specified number of instances using an AMI for which you have permissions.
When you launch an instance, it enters the pending state. After the instance is ready for you, it enters the running state. To check the state of your instance, call describe-instances .
If you don't specify a security group when launching an instance, Amazon EC2 uses the default security group. For more information, see Security Groups in the Amazon Elastic Compute Cloud User Guide .
Linux instances have access to the public key of the key pair at boot. You can use this key to provide secure access to the instance. Amazon EC2 public images use this feature to provide secure access without

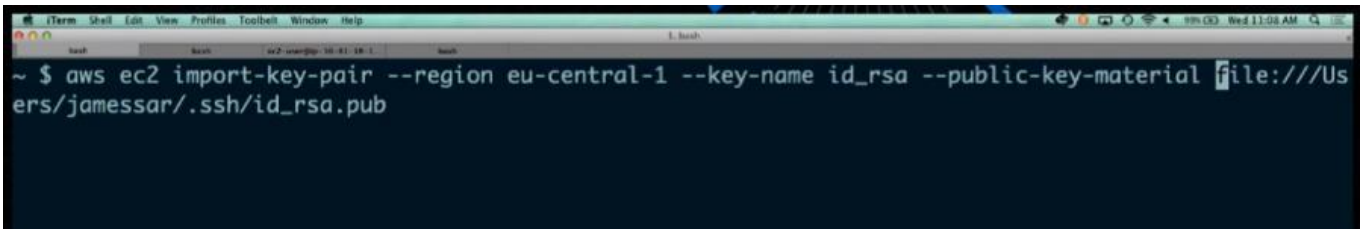

[--generate-cli-skeleton]
OPTIONS
--dry-run | --no-dry-run (boolean)
Checks whether you have the required permissions for the action, without actually making the request. Using this option will result in one of two possible error responses. If you have the required permissions, the error response will be DryRunOperation . Otherwise it will be UnauthorizedOperation .
--image-id (string)
The ID of the AMI, which you can get by calling describe-images .
--key-name (string)
The name of the key pair. You can create a key pair using create-key-pair or import-key-pair .
WARNING:
If you launch an instance without specifying a key pair, you can't connect to the instance.


--instance-type (string)
The instance type. For more information, see Instance Types in the Amazon Elastic Compute Cloud User Guide .
Default: m1.small
--placement (structure)
The placement for the instance.
Shorthand Syntax:
--placement AvailabilityZone=value,GroupName=value,Tenancy=value
JSON Syntax:
{
"AvailabilityZone": "string",
"GroupName": "string",
"Tenancy": "default"|"dedicated"
}


multiple times to assign multiple secondary IP addresses. If you want additional private IP addresses but do not need a specific address, use the --secondary-private-ip-address-count option.
--secondary-private-ip-address-count (string) [EC2-VPC] The number of secondary IP addresses to assign to the network interface or instance.
--associate-public-ip-address | --no-associate-public-ip-address (boolean) [EC2-VPC] If specified a public IP address will be assigned to the new instance in a VPC.
--cli-input-json (string) Performs service operation based on the JSON string provided. The JSON string follows the format provided by --generate-cli-skeleton. If other arguments are provided on the command line, it will not clobber their values.
--generate-cli-skeleton (boolean) Prints a sample input JSON to standard output. Note the specified operation is not run if this argument is specified. The sample input can be used as an argument for --cli-input-json.
EXAMPLES







The ***file:///…*** means that we are passing the content of the file specified as the value to use for the corresponding flag.

Let us now see some of the core scenarios most users fall into when using the AWS CLI


```
aws configure

AWS Access Key ID [**ABCD]:
AWS Secret Access Key [****************EFGH]:
Default region name [us-west-2]:
Default output format [None]:
```

```
aws configure <subcommand>

list - List common configuration sources
get  - Get the value of a single config var
set  - Set the value of a single config var
```

```
aws configure get region
```

The ***$ aws configure get region*** command above is going to print out the currently configured region that we have on our computer's stdout.

```
aws configure set profile.prod.region us-west-2
```

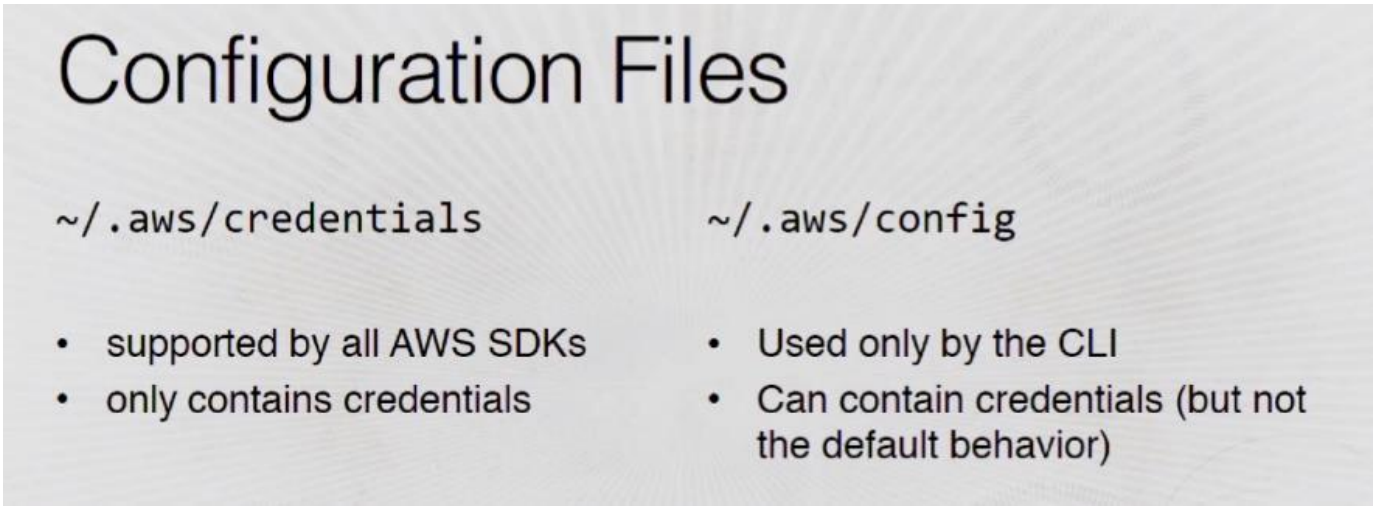
We can also set values using the command pattern like ***$ aws configure set profile.prod.region.region us-west-2*** to set the region for the prod profile.

A profile is a group of configuration values

We might have a ***dev*** profile that has credentials to access specific resources in our AWS account, and we might also have a ***prod*** profile with access to production related resources in our account.

```
aws configure --profile prod
```

The ***$ aws configure - -profile prod*** command will create a new prod profile in our configuration file. We can then start appending the ***- -profile prod*** to commands we use to have that command run with the profile credentials and access permissions that we set up.



The AWS CLI will normally write out your credentials to 2 separate files above, there is a ***credentials file ~/.aws/credentials*** is shared and only contains credentials like the ***accessKey***, ***secretKey***, and an optional ***sessionToken***. By default, the AWS CLI will write all the credentials to the shared credentials file because other services understand this format. The ***config file ~/.aws/config*** is only used by the CLI and contains CLI specific things like what output format to use, what region to use.

```
aws configure set profile.prod.aws_access_key_id foo
```

~/.aws/credentials

```
[prod]
aws_access_key_id = foo
```

~/.aws/config

```
aws configure set profile.prod.aws_secret_access_key bar
```

~/.aws/credentials

```
[prod]
aws_access_key_id = foo
aws_secret_access_key = bar
```

~/.aws/config

```
aws configure set profile.prod.region us-west-2
```

~/.aws/credentials

```
[prod]
aws_access_key_id = foo
aws_secret_access_key = bar
```

~/.aws/config

```
[profile prod]
region = us-west-2
```

```
aws configure set profile.prod.output text
```

~/.aws/credentials

```
[prod]
aws_access_key_id = foo
aws_secret_access_key = bar
```

~/.aws/config

```
[profile prod]
region = us-west-2
output = text
```

create-new-user.sh

```
#!/bin/bash
# Create a new user and create a new profile.
aws iam create-user --user-name reinvent-user
credentials=$(aws iam create-access-key --user-name reinvent-user \
    --query 'AccessKey.[AccessKeyId,SecretAccessKey]' \
    --output text)
access_key_id=$(echo $credentials | cut -d' ' -f 1)
secret_access_key=$(echo $credentials | cut -d' ' -f 2)
aws configure set profile.reinvent.aws_access_key_id "$access_key_id"
aws configure set profile.reinvent.secret_access_key "$secret_access_key"
```

This is an example where we create an IAM user, create the credentials, and have it configured with the new profile so that we can use it in the AWS CLI.

create-new-user.sh

```
#!/bin/bash
# Create a new user and create a new profile.
aws iam create-user --user-name reinvent-user
credentials=$(aws iam create-access-key --user-name reinvent-user \
    --query 'AccessKey.[AccessKeyId,SecretAccessKey]' \
    --output text)
access_key_id=$(echo $credentials | cut -d' ' -f 1)
secret_access_key=$(echo $credentials | cut -d' ' -f 2)
aws configure set profile.reinvent.aws_access_key_id "$access_key_id"
aws configure set profile.reinvent.secret_access_key "$secret_access_key"
```


We are going to first create a user called reinvent-user using the command ***$ aws iam create-user - -user-name reinvent-user.*** There are no credentials created for a new user by default.

create-new-user.sh

```
#!/bin/bash
# Create a new user and create a new profile.
aws iam create-user --user-name reinvent-user
credentials=$(aws iam create-access-key --user-name reinvent-user \
    --query 'AccessKey.[AccessKeyId,SecretAccessKey]' \
    --output text)
access_key_id=$(echo $credentials | cut -d' ' -f 1)
secret_access_key=$(echo $credentials | cut -d' ' -f 2)
aws configure set profile.reinvent.aws_access_key_id "$access_key_id"
aws configure set profile.reinvent.secret_access_key "$secret_access_key"
```


We then create credentials for this user using the command ***$ credentials=$(aws iam create-access-key - -user-name reinvent-user - -query 'AccessKey.[AccessKeyId,SecretAccessKey]' - -output text)***

create-new-user.sh

```
#!/bin/bash
# Create a new user and create a new profile.
aws iam create-user --user-name reinvent-user
credentials=$(aws iam create-access-key --user-name reinvent-user \
    --query 'AccessKey.[AccessKeyId,SecretAccessKey]' \
    --output text)
access_key_id=$(echo $credentials | cut -d' ' -f 1)
secret_access_key=$(echo $credentials | cut -d' ' -f 2)
aws configure set profile.reinvent.aws_access_key_id "$access_key_id"
aws configure set profile.reinvent.secret_access_key "$secret_access_key"
```

We then grab the ***AccessKey*** using ***$ access_key_id=$(echo $credentials | cut -d' ' -f 1)*** and also grab the ***SecretKey*** using ***$ secret_access_key=$(echo $credentials | cut -d' ' -f 2)***

create-new-user.sh

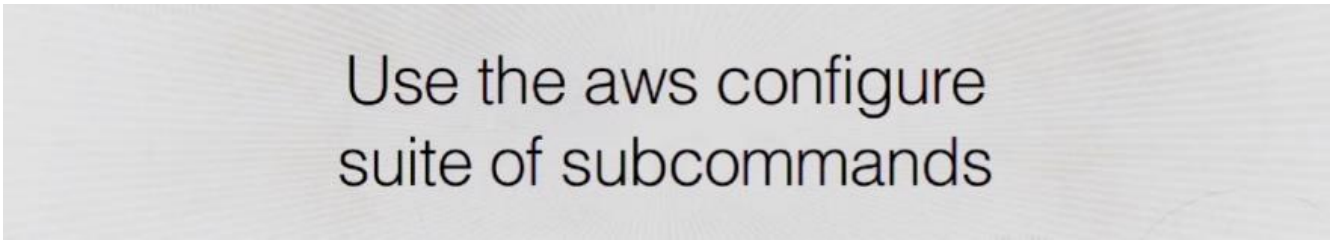
```
#!/bin/bash
# Create a new user and create a new profile.
aws iam create-user --user-name reinvent-user
credentials=$(aws iam create-access-key --user-name reinvent-user \
    --query 'AccessKey.[AccessKeyId,SecretAccessKey]' \
    --output text)
access_key_id=$(echo $credentials | cut -d' ' -f 1)
secret_access_key=$(echo $credentials | cut -d' ' -f 2)
aws configure set profile.reinvent.aws_access_key_id "$access_key_id"
aws configure set profile.reinvent.secret_access_key "$secret_access_key"
```

We then use the ***set*** command to create the new profile for the user using the ***AccessKey*** and the ***SecretKey*** with ***$ aws configure set profile.reinvent.aws_access_key_id "$access_key_id"*** and ***$ aws configure set profile.reinvent.secret_access_key "$secret_access_key"***

create-new-user.sh

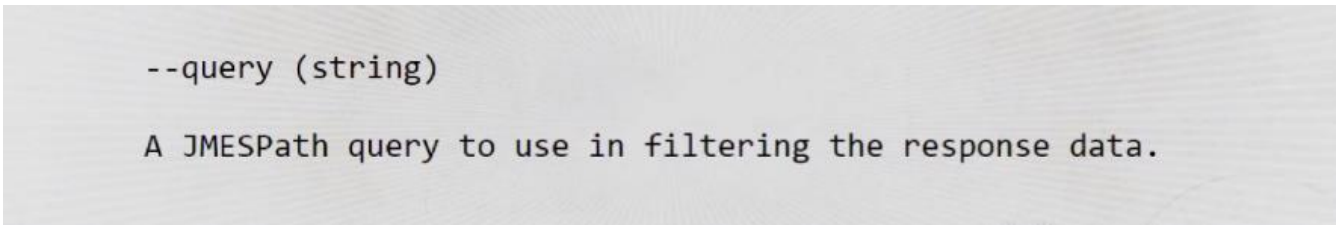
```
#!/bin/bash
# Create a new user and create a new profile.
aws iam create-user --user-name reinvent-user
credentials=$(aws iam create-access-key --user-name reinvent-user \
    --query 'AccessKey.[AccessKeyId,SecretAccessKey]' \
    --output text)
access_key_id=$(echo $credentials | cut -d' ' -f 1)
secret_access_key=$(echo $credentials | cut -d' ' -f 2)
aws configure set profile.reinvent.aws_access_key_id "$access_key_id"
aws configure set profile.reinvent.secret_access_key "$secret_access_key"
```

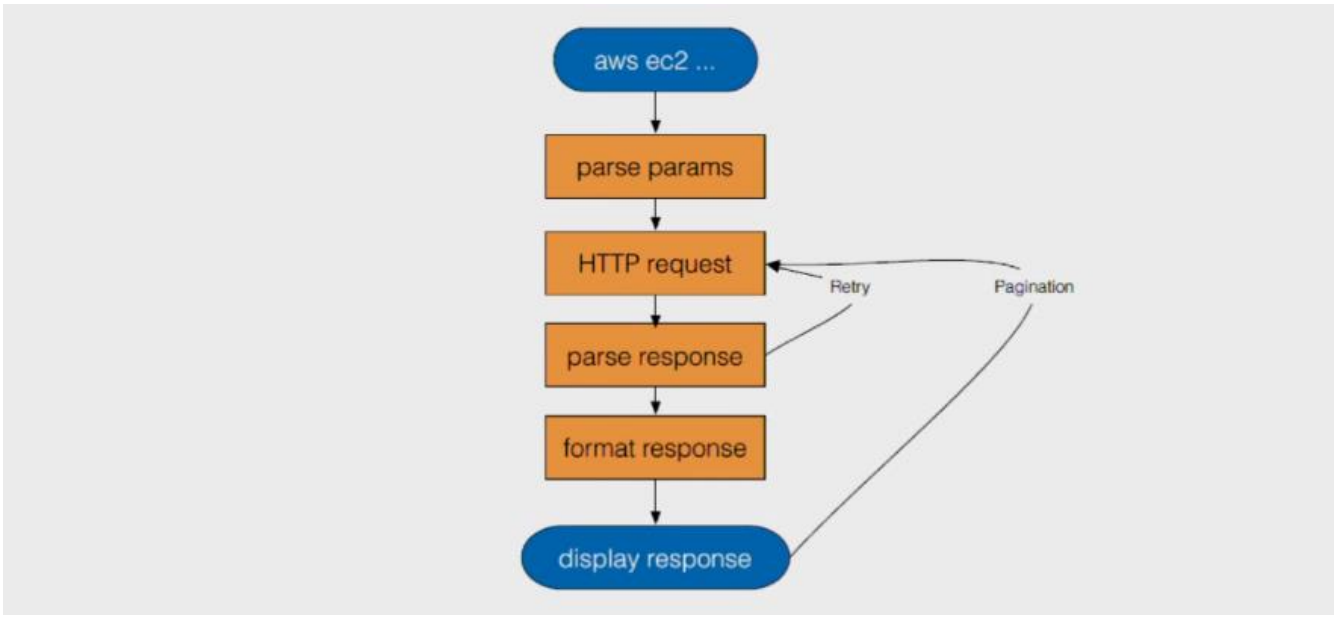
This allows you to create a new profile within the CLI entirely programmatically

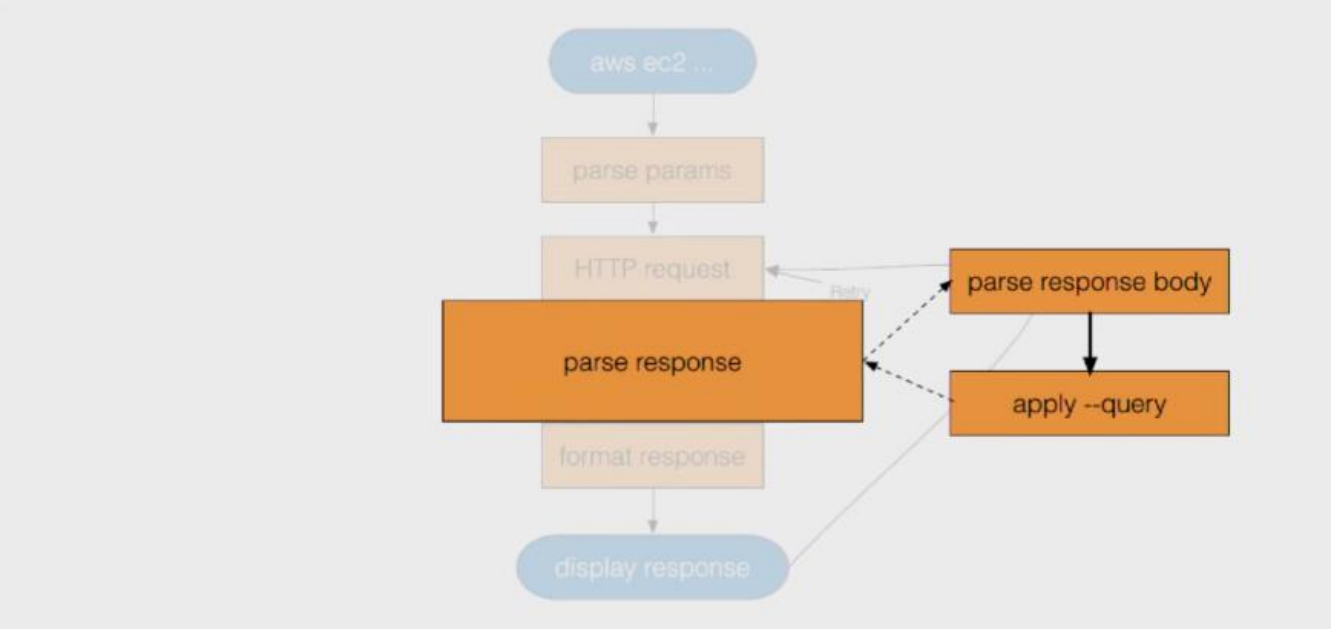


This gives you programmatic access to your configuration values, you should check out the aws configure suite of subcommands.

One of the most common thing to do in the CLI is to run a command and then get a response back but you only want a subset of that data. this is where you can use the ***- -query*** expression option to filter the results of that command.








```
<ListUsersResponse xmlns="...">
  <ListUsersResult>
    <Users>
      <member>
        <UserId>userid</UserId>
        <Path>/</Path>
        <UserName>james</UserName>
        <Arn>arn:aws:iam::..:user/james</Arn>
        <CreateDate>2013-03-09T23:36:32Z</CreateDate>
      </member>
    <Users>
  </ListUsersResult>
<ListUsersResponse>
```

Imagine we have this AWS service response that has this XML body,

```
{
  "Users": [
    {
      "Arn": "arn:aws:iam::...:user/james",
      "UserId": "userid",
      "CreateDate": "2013-03-09T23:36:32Z",
      "Path": "/",
      "UserName": "james"
    }
  ]
}
```

We normally parse it into some Python data structure like a list or dictionary as above

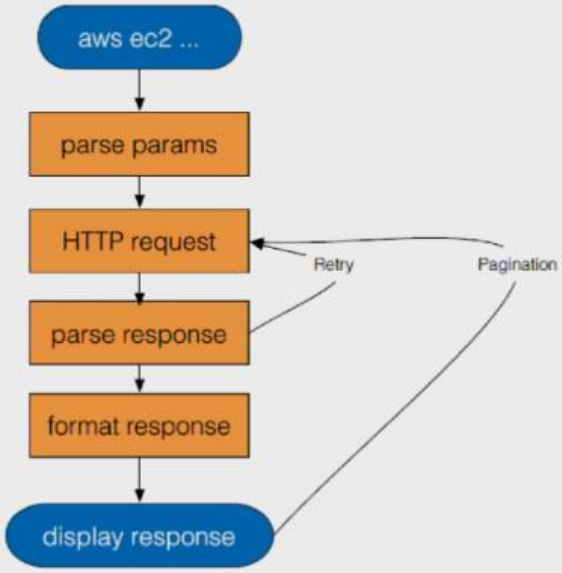
```
--query Users[0].[UserName,Path,UserId]

  {
    "Users": [
      {
        "Arn": "arn:aws:iam::...:user/james",
        "UserId": "userid",
        "CreateDate": "2013-03-09T23:36:32Z",
        "Path": "/",
        "UserName": "james"
      }
    ]
  }
```

We then see if the query has been applied

```
--query Users[0].[UserName,Path,UserId]


        [
            [
                "james", "/", "id"
            ],
        ]
```

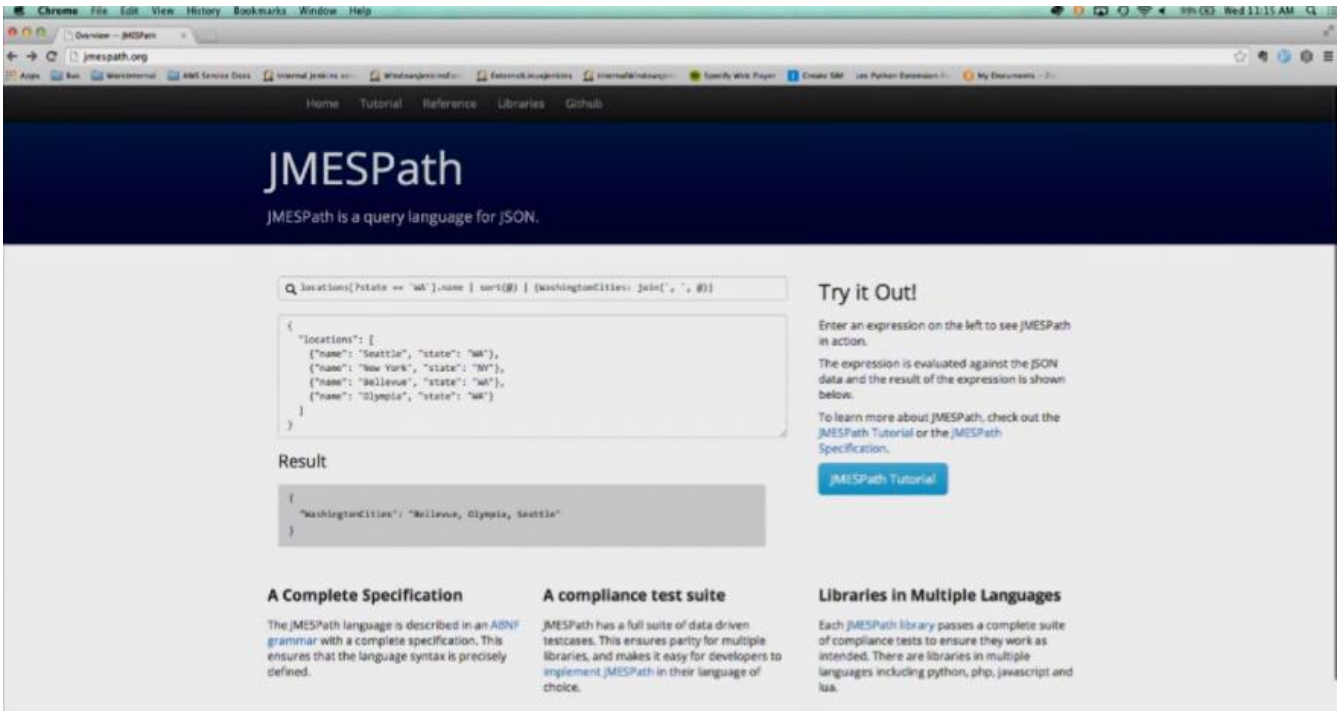
Then we further filter the data down to exactly what the query expression wanted

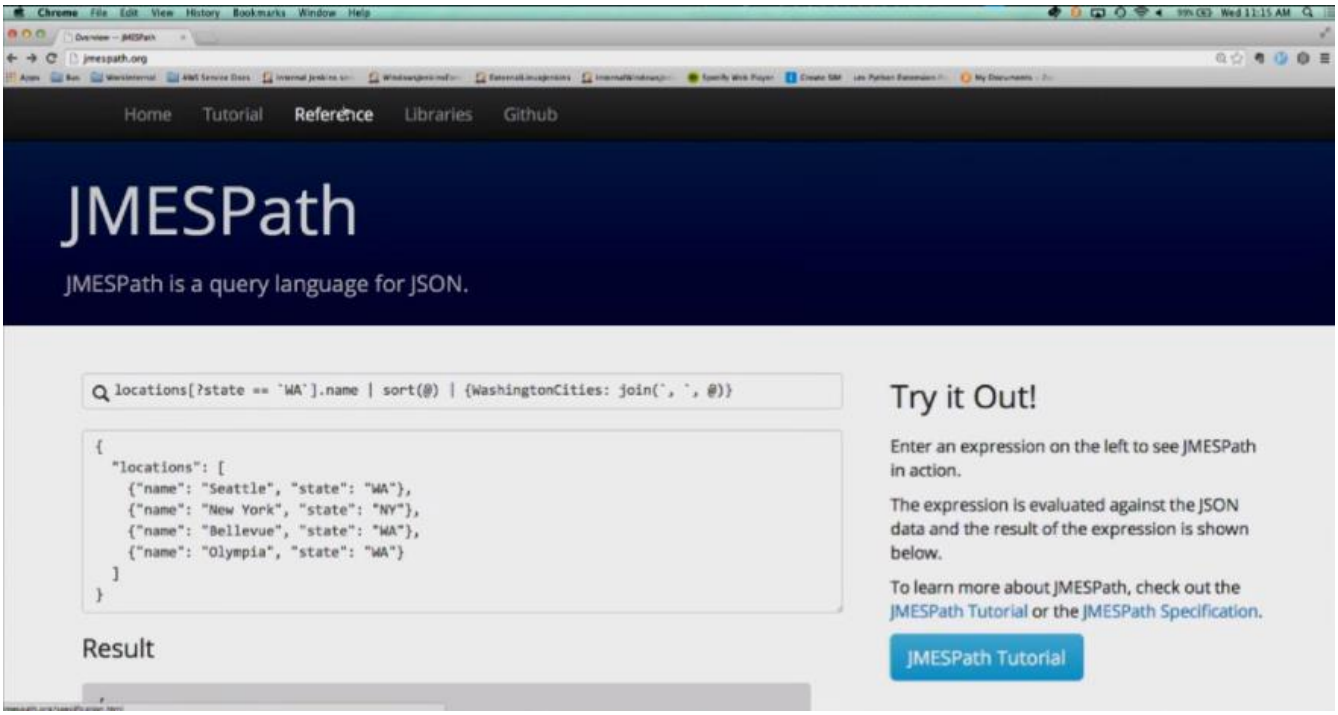
```
-----------------------------------------
|               ListUsers               |
+------------------------+----+----------+
|  james                 |  / |  userid  |
+------------------------+----+----------+
```

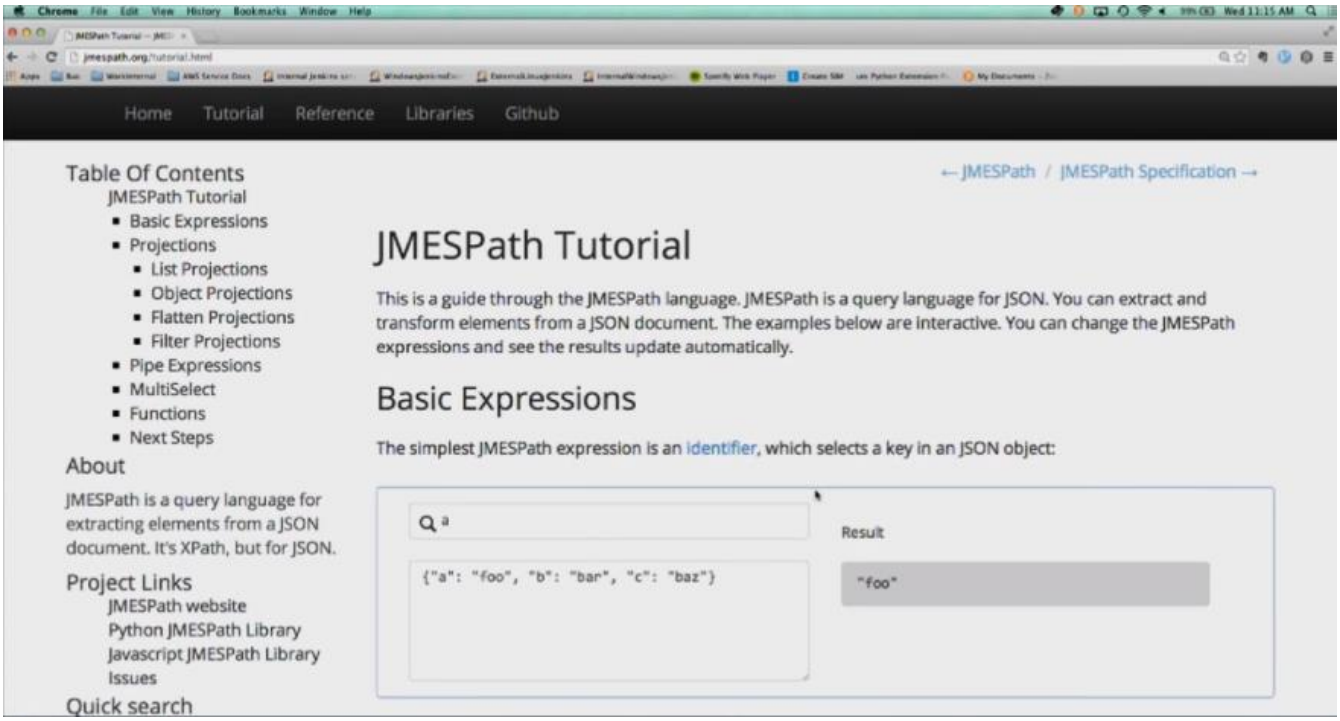


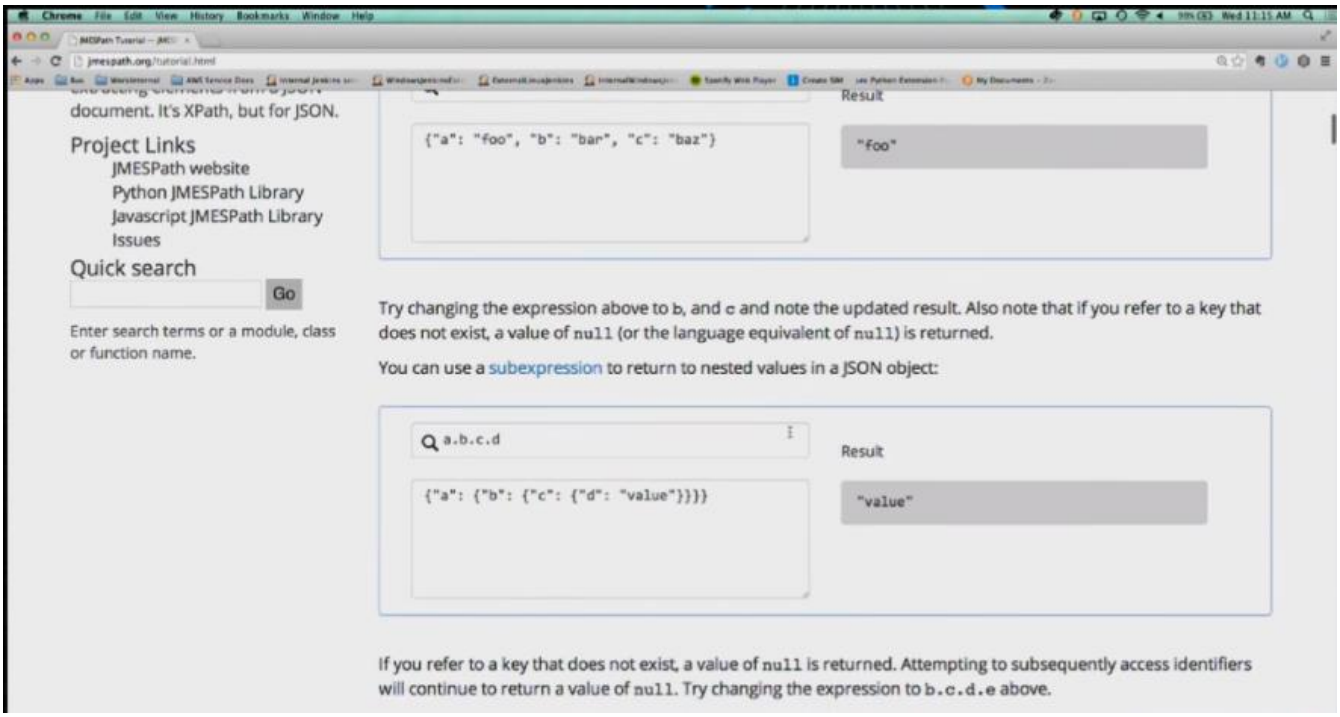
http://jmespath.org

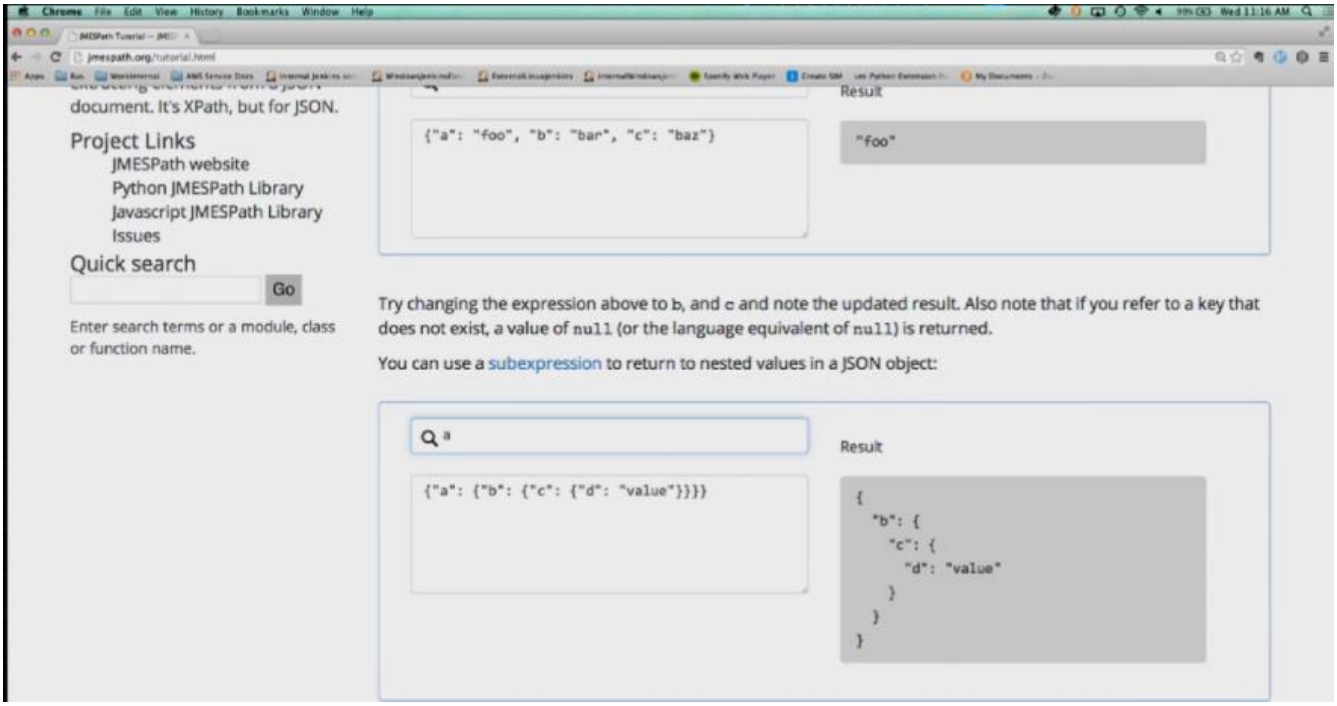
A Query Language For JSON

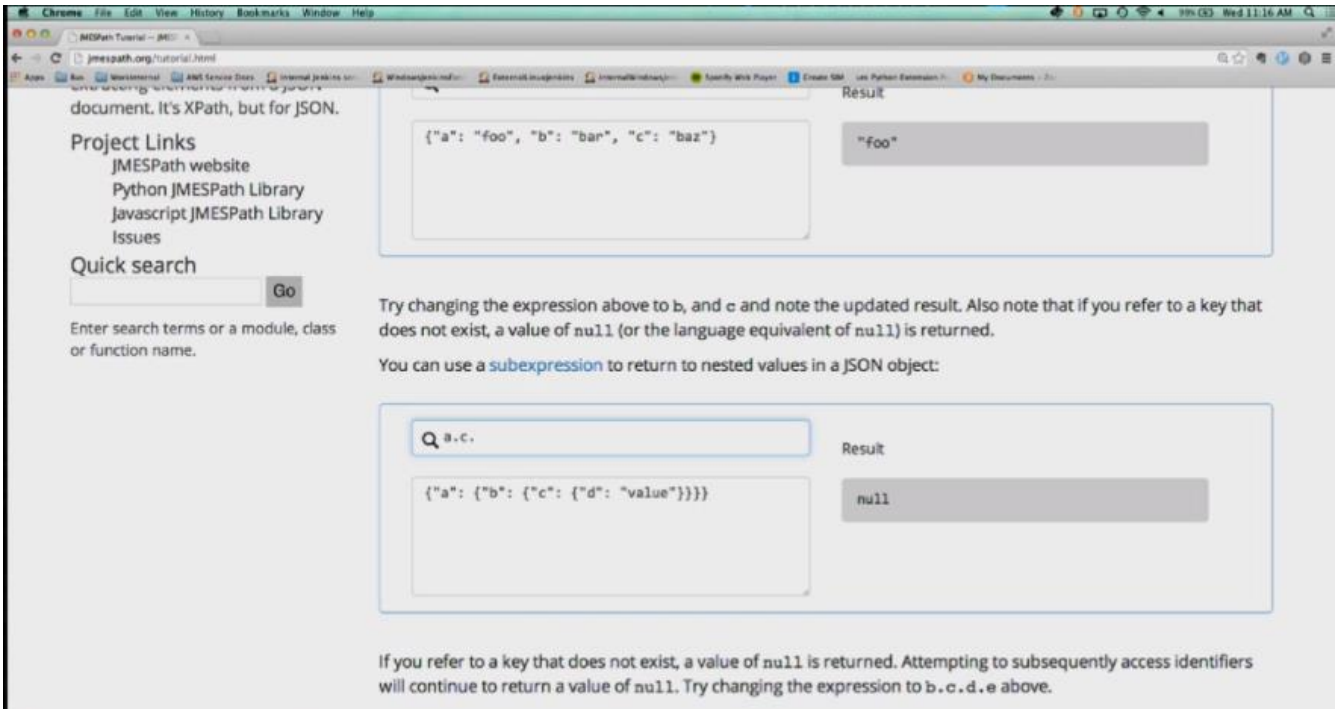


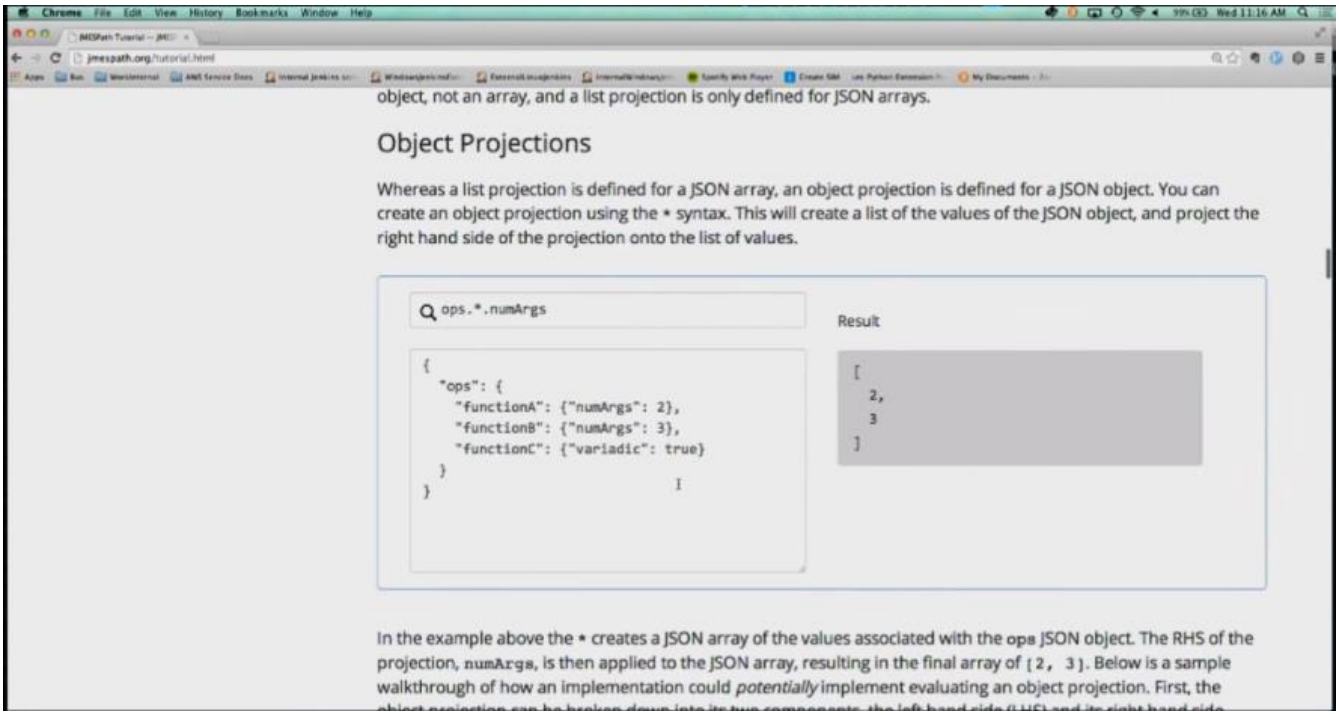


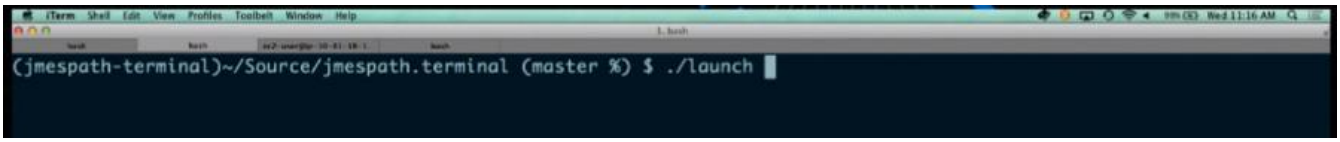

Home Tutorial Reference Libraries Github
Table Of Contents
JMESPath Tutorial
Basic Expressions
Projections
List Projections
Object Projections
Flatten Projections
Filter Projections
Pipe Expressions
MultiSelect
Functions
Next Steps
About
JMESPath is a query language for extracting elements from a JSON document. It's XPath, but for JSON.
Project Links
JMESPath website
Python JMESPath Library
Javascript JMESPath Library
Issues
Quick search
← JMESPath / JMESPath Specification →
JMESPath Tutorial
This is a guide through the JMESPath language. JMESPath is a query language for JSON. You can extract and transform elements from a JSON document. The examples below are interactive. You can change the JMESPath expressions and see the results update automatically.
Basic Expressions
The simplest JMESPath expression is an identifier, which selects a key in an JSON object:
a
{"a": "foo", "b": "bar", "c": "baz"}
Result
"foo"


document. It's XPath, but for JSON.
Project Links
JMESPath website
Python JMESPath Library
Javascript JMESPath Library
Issues
Quick search
Go
Enter search terms or a module, class or function name.
{"a": "foo", "b": "bar", "c": "baz"}
Result
"foo"
Try changing the expression above to b, and c and note the updated result. Also note that if you refer to a key that does not exist, a value of null (or the language equivalent of null) is returned.
You can use a subexpression to return to nested values in a JSON object:
a.b.c.d
{"a": {"b": {"c": {"d": "value"}}}}
Result
"value"
If you refer to a key that does not exist, a value of null is returned. Attempting to subsequently access identifiers will continue to return a value of null. Try changing the expression to b.c.d.e above.

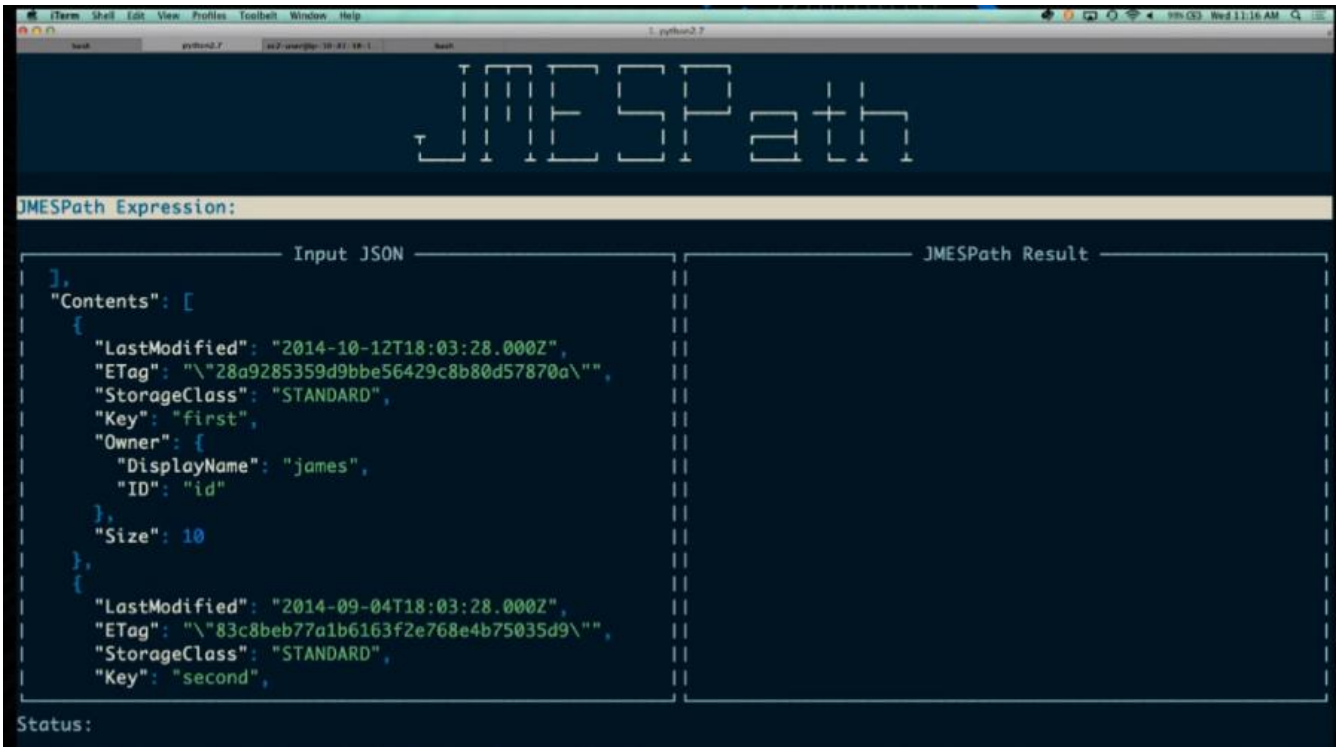

document. It's XPath, but for JSON.
Project Links
JMESPath website
Python JMESPath Library
Javascript JMESPath Library
Issues
Quick search
Go
Enter search terms or a module, class or function name.
Result
{"a": "foo", "b": "bar", "c": "baz"}
"foo"
Try changing the expression above to b, and c and note the updated result. Also note that if you refer to a key that does not exist, a value of null (or the language equivalent of null) is returned.
You can use a subexpression to return to nested values in a JSON object:
a
{"a": {"b": {"c": {"d": "value"}}}}
Result
{
"b": {
"c": {
"d": "value"
}
}
}


document. It's XPath, but for JSON.
Project Links
JMESPath website
Python JMESPath Library
Javascript JMESPath Library
Issues
Quick search
Go
Enter search terms or a module, class or function name.
Result
{"a": "foo", "b": "bar", "c": "baz"}
"foo"
Try changing the expression above to b, and c and note the updated result. Also note that if you refer to a key that does not exist, a value of null (or the language equivalent of null) is returned.
You can use a subexpression to return to nested values in a JSON object:
a.c.
{"a": {"b": {"c": {"d": "value"}}}}
Result
null
If you refer to a key that does not exist, a value of null is returned. Attempting to subsequently access identifiers will continue to return a value of null. Try changing the expression to b.c.d.e above.

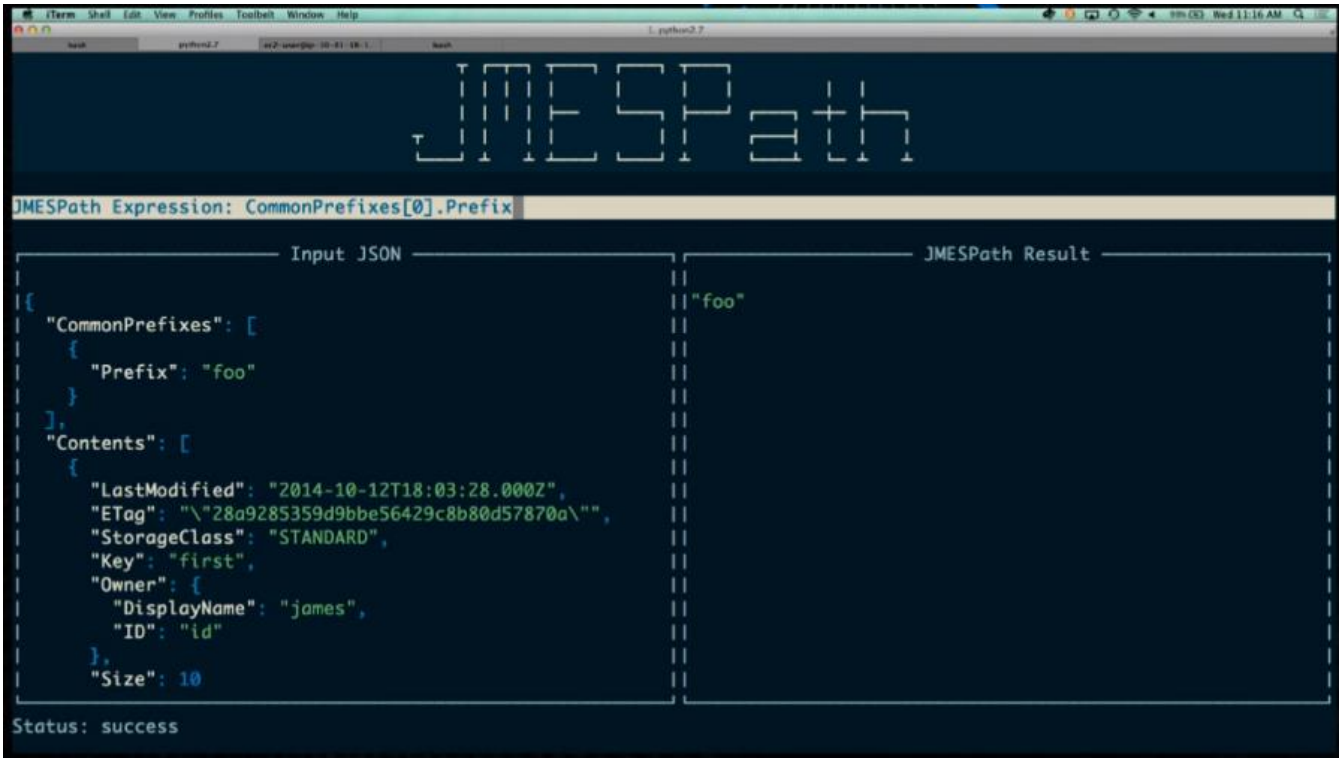

object, not an array, and a list projection is only defined for JSON arrays.
Object Projections
Whereas a list projection is defined for a JSON array, an object projection is defined for a JSON object. You can create an object projection using the * syntax. This will create a list of the values of the JSON object, and project the right hand side of the projection onto the list of values.
ops.*.numArgs
{
"ops": {
"functionA": {"numArgs": 2},
"functionB": {"numArgs": 3},
"functionC": {"variadic": true}
}
}
Result
[
2,
3
]
In the example above the * creates a JSON array of the values associated with the ops JSON object. The RHS of the projection, numArgs, is then applied to the JSON array, resulting in the final array of [2, 3]. Below is a sample walkthrough of how an implementation could potentially implement evaluating an object projection. First, the


(jmespath-terminal)~/Source/jmespath.terminal (master %) $ ./launch

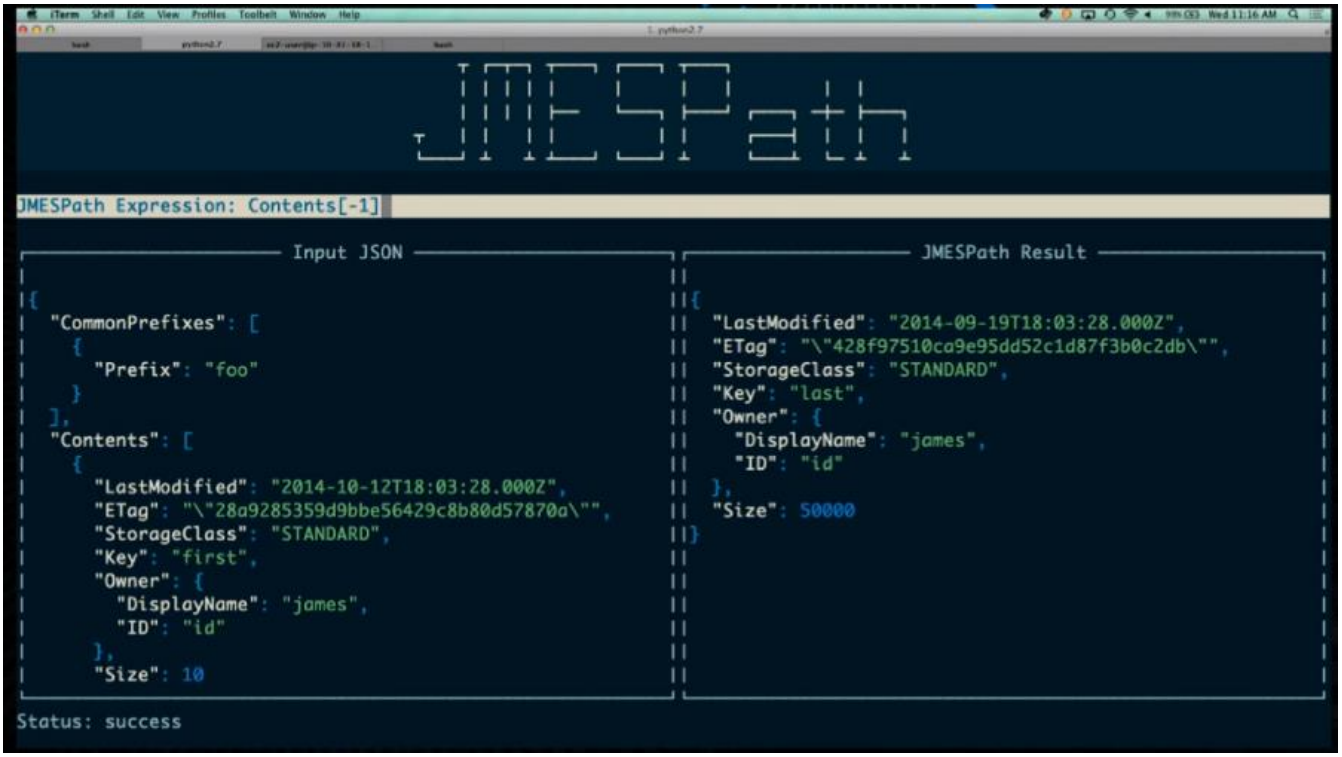

JMESPath Expression:
Input JSON
JMESPath Result
{
"CommonPrefixes": [
{
"Prefix": "foo"
}
],
"Contents": [
{
"LastModified": "2014-10-12T18:03:28.000Z",
"ETag": "\"28a9285359d9bbe56429c8b80d57870a\"",
"StorageClass": "STANDARD",
"Key": "first",
"Owner": {
"DisplayName": "james",
"ID": "id"
},
"Size": 10
Status:



We have some sample input on the right side


JMESPath
JMESPath Expression: CommonPrefixes[0].Prefix
Input JSON
{
"CommonPrefixes": [
{
"Prefix": "foo"
}
],
"Contents": [
{
"LastModified": "2014-10-12T18:03:28.000Z",
"ETag": "\"28a9285359d9bbe56429c8b80d57870a\"",
"StorageClass": "STANDARD",
"Key": "first",
"Owner": {
"DisplayName": "james",
"ID": "id"
},
"Size": 10
JMESPath Result
"foo"
Status: success

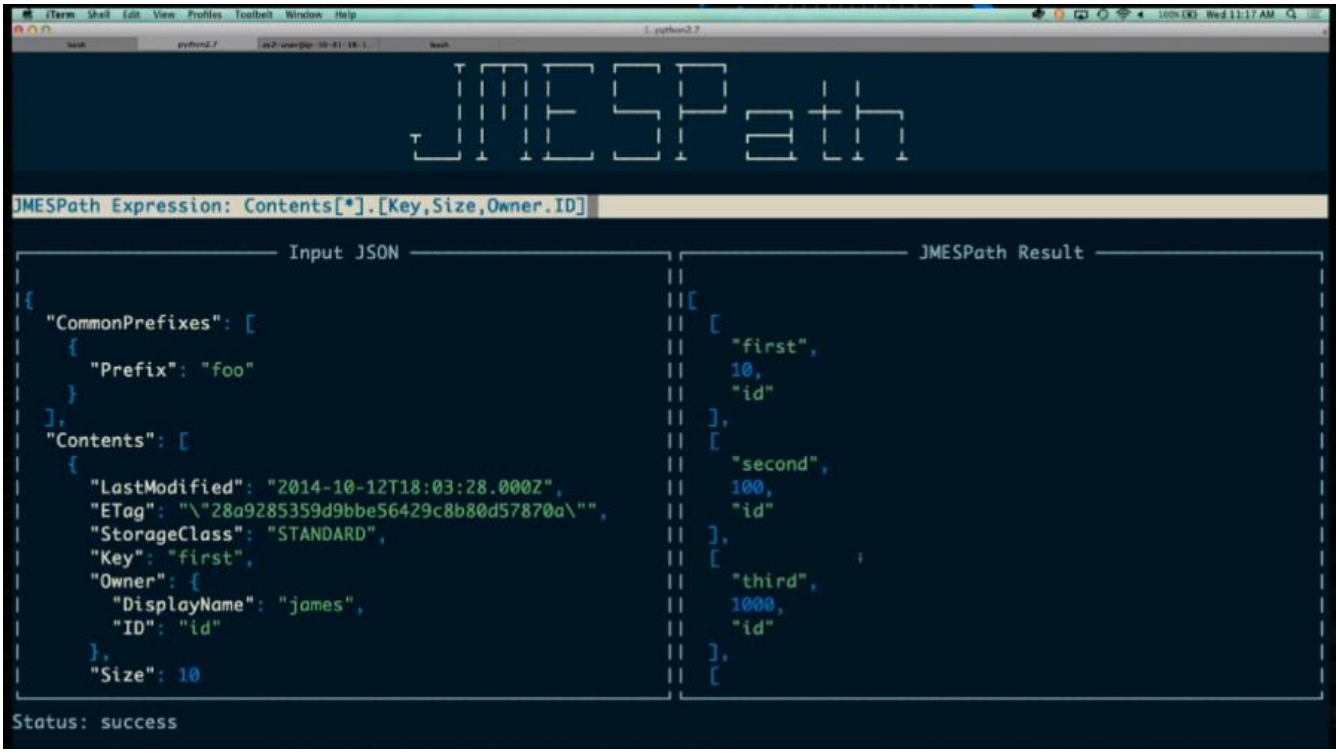

JMESPath
JMESPath Expression: Contents[0]
Input JSON
{
"CommonPrefixes": [
{
"Prefix": "foo"
}
],
"Contents": [
{
"LastModified": "2014-10-12T18:03:28.000Z",
"ETag": "\"28a9285359d9bbe56429c8b80d57870a\"",
"StorageClass": "STANDARD",
"Key": "first",
"Owner": {
"DisplayName": "james",
"ID": "id"
},
"Size": 10
JMESPath Result
{
"LastModified": "2014-10-12T18:03:28.000Z",
"ETag": "\"28a9285359d9bbe56429c8b80d57870a\"",
"StorageClass": "STANDARD",
"Key": "first",
"Owner": {
"DisplayName": "james",
"ID": "id"
},
"Size": 10
}
Status: success


JMESPath Expression: Contents[-1]
Input JSON
{
"CommonPrefixes": [
{
"Prefix": "foo"
}
],
"Contents": [
{
"LastModified": "2014-10-12T18:03:28.000Z",
"ETag": "\"28a9285359d9bbe56429c8b80d57870a\"",
"StorageClass": "STANDARD",
"Key": "first",
"Owner": {
"DisplayName": "james",
"ID": "id"
},
"Size": 10
JMESPath Result
{
"LastModified": "2014-09-19T18:03:28.000Z",
"ETag": "\"428f97510ca9e95dd52c1d87f3b0c2db\"",
"StorageClass": "STANDARD",
"Key": "last",
"Owner": {
"DisplayName": "james",
"ID": "id"
},
"Size": 50000
}
Status: success


JMESPath Expression: Contents[-2]
Input JSON
{
"CommonPrefixes": [
{
"Prefix": "foo"
}
],
"Contents": [
{
"LastModified": "2014-10-12T18:03:28.000Z",
"ETag": "\"28a9285359d9bbe56429c8b80d57870a\"",
"StorageClass": "STANDARD",
"Key": "first",
"Owner": {
"DisplayName": "james",
"ID": "id"
},
"Size": 10
JMESPath Result
{
"LastModified": "2014-09-04T18:03:28.000Z",
"ETag": "\"ab5a585cca8fd488ea9a7ed31215d550\"",
"StorageClass": "STANDARD",
"Key": "next-to-last",
"Owner": {
"DisplayName": "james",
"ID": "id"
},
"Size": 10000
}
Status: success

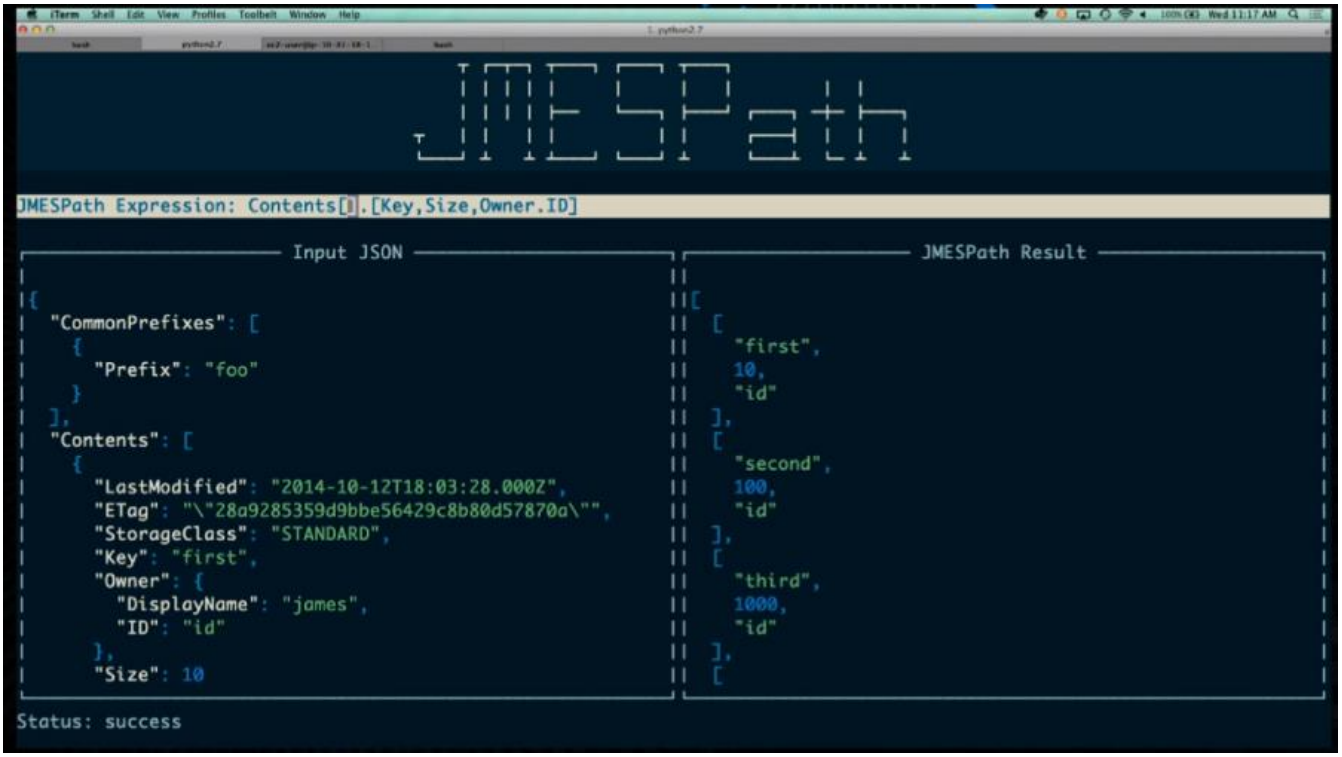

JMESPath Expression: Contents[*]
Input JSON
JMESPath Result
{
"CommonPrefixes": [
{
"Prefix": "foo"
}
],
"Contents": [
{
"LastModified": "2014-10-12T18:03:28.000Z",
"ETag": "\"28a9285359d9bbe56429c8b80d57870a\"",
"StorageClass": "STANDARD",
"Key": "first",
"Owner": {
"DisplayName": "james",
"ID": "id"
},
"Size": 10
[
{
"LastModified": "2014-10-12T18:03:28.000Z",
"ETag": "\"28a9285359d9bbe56429c8b80d57870a\"",
"StorageClass": "STANDARD",
"Key": "first",
"Owner": {
"DisplayName": "james",
"ID": "id"
},
"Size": 10
},
{
"LastModified": "2014-09-04T18:03:28.000Z",
"ETag": "\"83c8beb77a1b6163f2e768e4b75035d9\"",
"StorageClass": "STANDARD",
"Key": "second",
Status: success


JMESPath Expression: Contents[*].Key
Input JSON
JMESPath Result
{
"CommonPrefixes": [
{
"Prefix": "foo"
}
],
"Contents": [
{
"LastModified": "2014-10-12T18:03:28.000Z",
"ETag": "\"28a9285359d9bbe56429c8b80d57870a\"",
"StorageClass": "STANDARD",
"Key": "first",
"Owner": {
"DisplayName": "james",
"ID": "id"
},
"Size": 10
[
"first",
"second",
"third",
"fourth",
"fifth",
"next-to-last",
"last"
]
Status: success



This is how we can get a list of lists

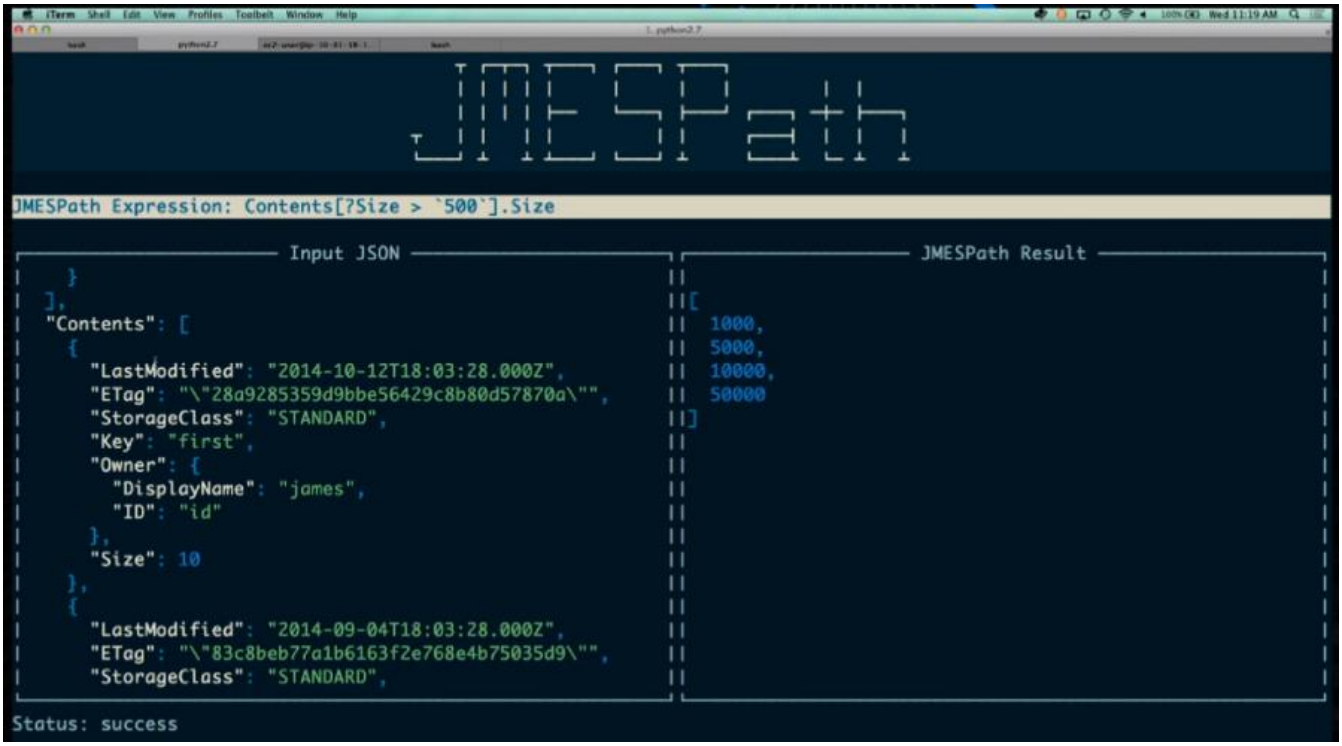
This is how we can take down the list of lists by one level using flattening

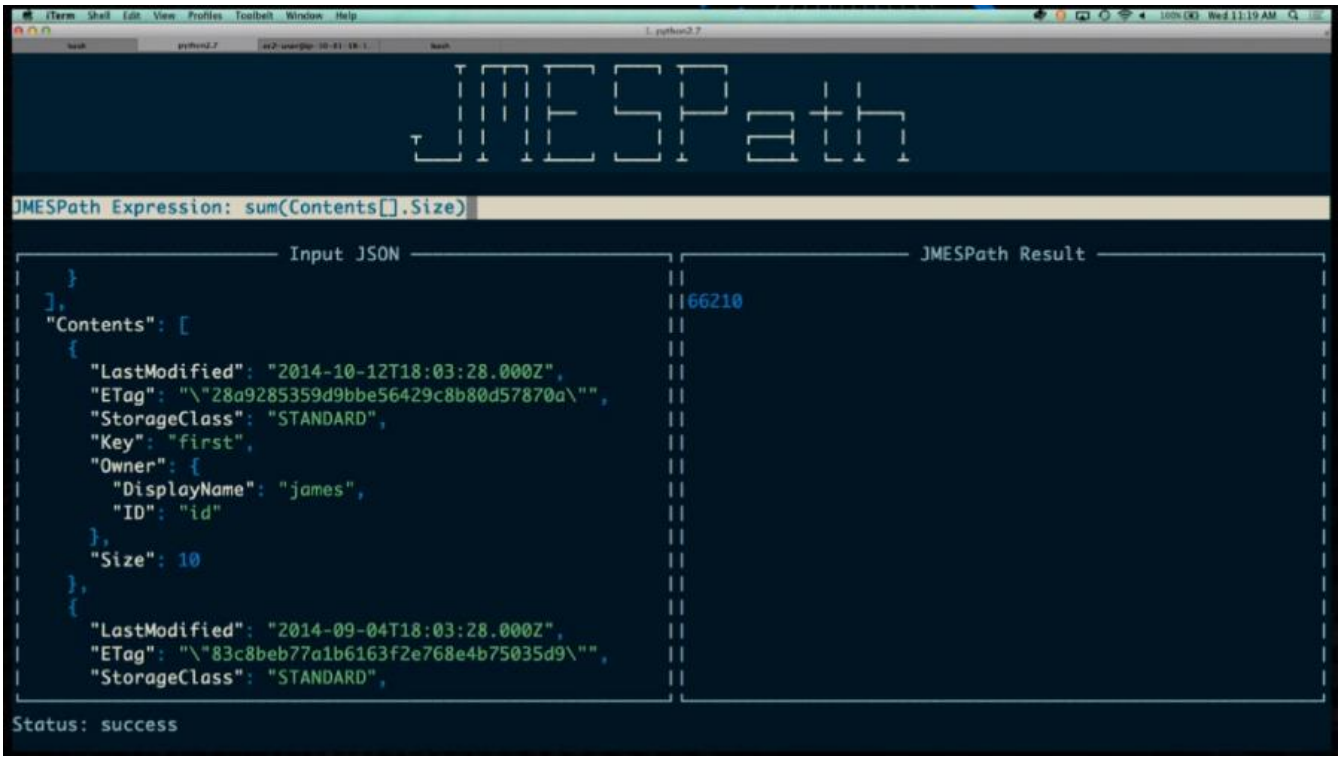

JMESPath
JMESPath Expression: Contents[].[Key,Size,Owner.ID]
Input JSON
{
"CommonPrefixes": [
{
"Prefix": "foo"
}
],
"Contents": [
{
"LastModified": "2014-10-12T18:03:28.000Z",
"ETag": "\"28a9285359d9bbe56429c8b80d57870a\"",
"StorageClass": "STANDARD",
"Key": "first",
"Owner": {
"DisplayName": "james",
"ID": "id"
},
"Size": 10
JMESPath Result
[
[
"first",
10,
"id"
],
[
"second",
100,
"id"
],
[
"third",
1000,
"id"
],
[
Status: success


JMESPath
JMESPath Expression: Contents[].{Key: Key, Bytes: Size, ID: Owner.ID}
Input JSON
{
"CommonPrefixes": [
{
"Prefix": "foo"
}
],
"Contents": [
{
"LastModified": "2014-10-12T18:03:28.000Z",
"ETag": "\"28a9285359d9bbe56429c8b80d57870a\"",
"StorageClass": "STANDARD",
"Key": "first",
"Owner": {
"DisplayName": "james",
"ID": "id"
},
"Size": 10
JMESPath Result
[
{
"Bytes": 10,
"ID": "id",
"Key": "first"
},
{
"Bytes": 100,
"ID": "id",
"Key": "second"
},
{
"Bytes": 1000,
"ID": "id",
"Key": "third"
},
{
Status: success

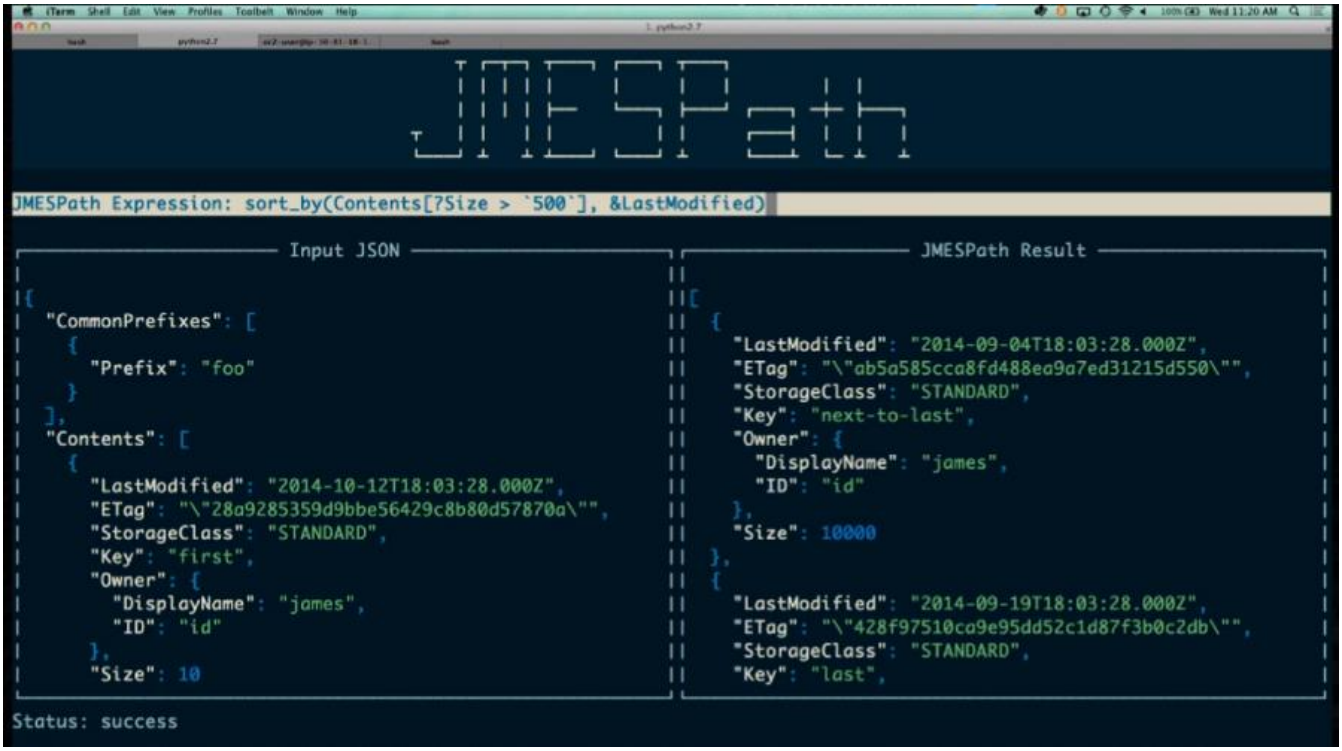
We can also specify literal strings like the ***`S3Object`*** used above

We have also added the ability to filter down a list using query expressions,

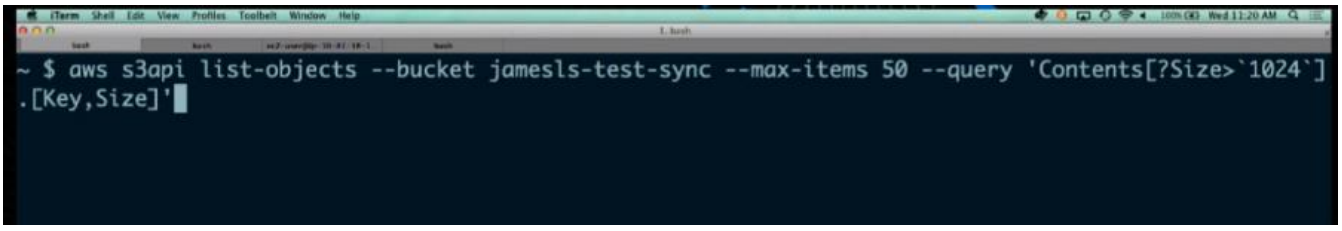




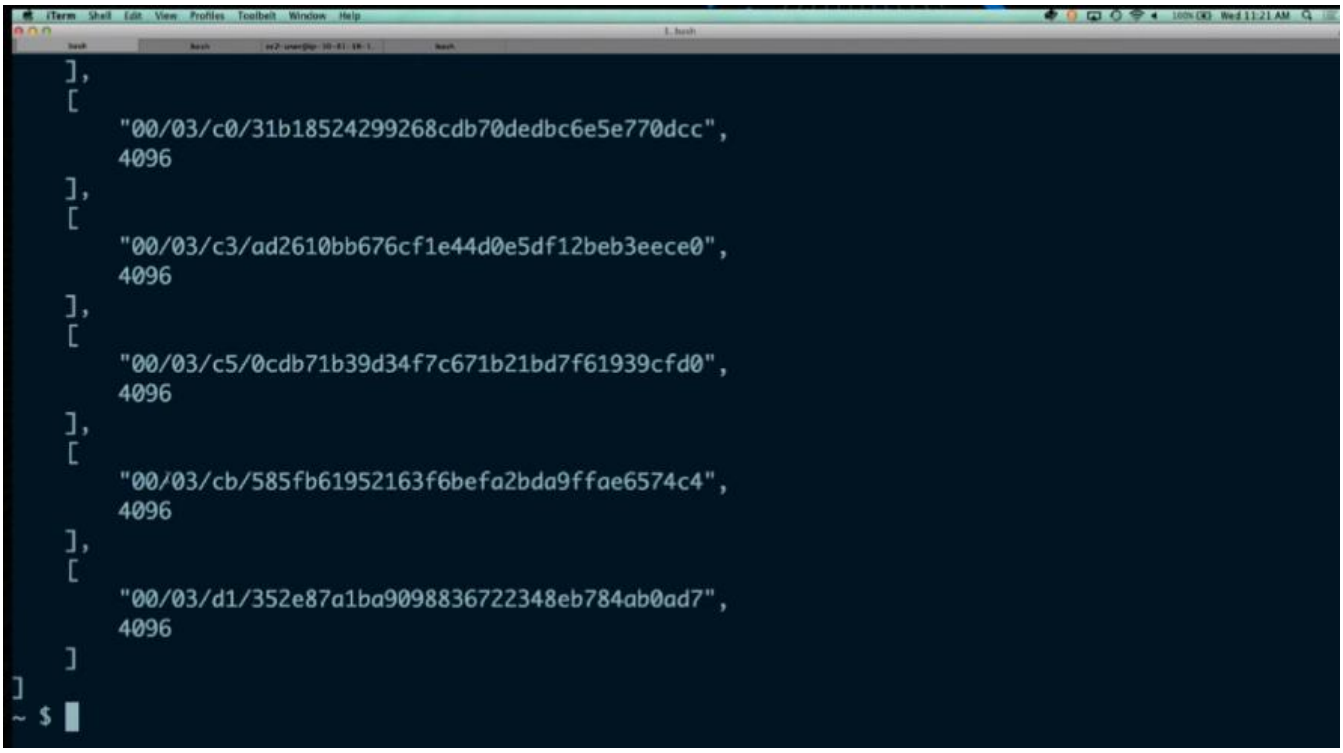
We have also added the ability to use functions within our query expression as above

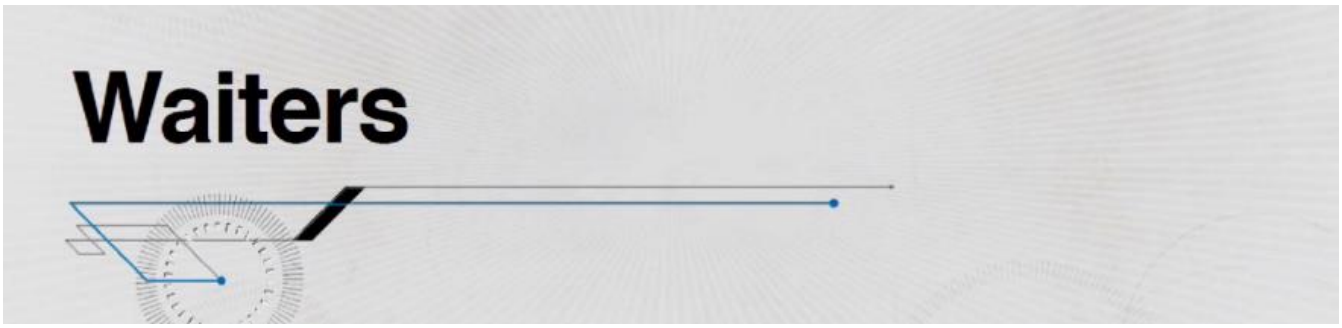


We also have the ability to sort the response object that we get back from a query expression as above

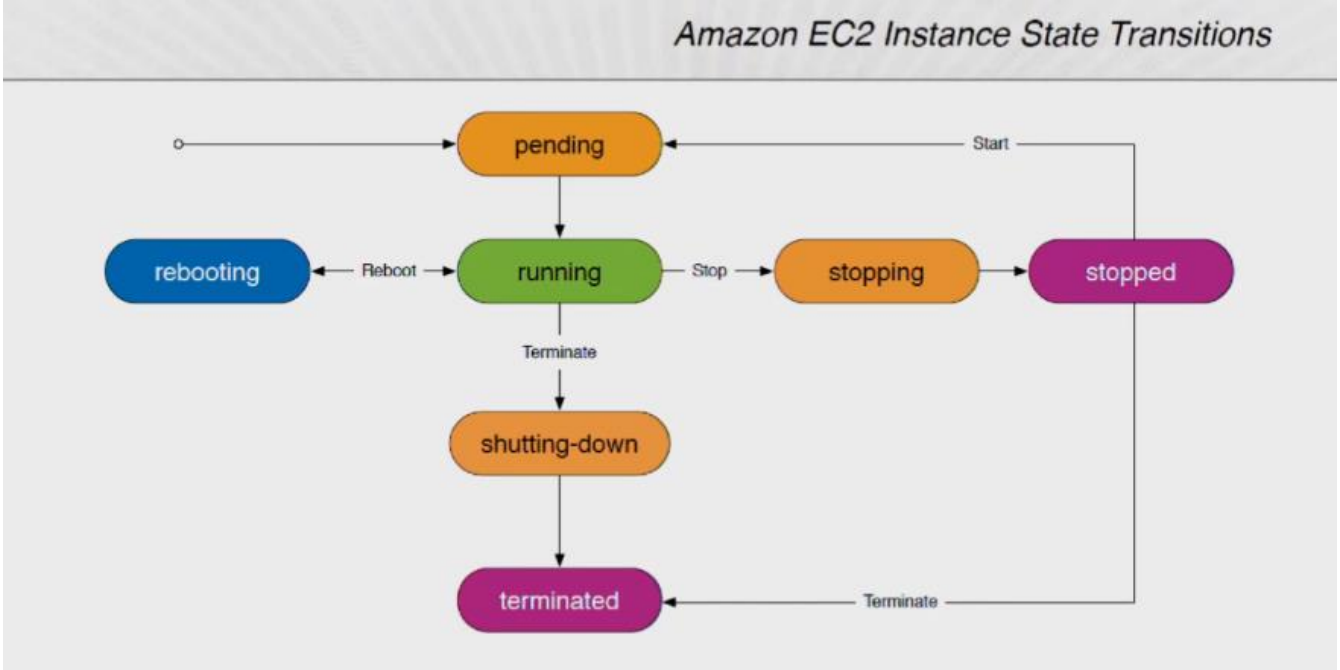


We are grabbing the Key and Size only from the response object for this query expression





One of the most common things you do in the CLI in addition to issuing commands, querying and filtering down your outputs is to invoke some sort of command and then polling and waiting for it to reach a particular state



An example is launching an EC2 instance, it goes into a ***pending*** state as soon as we launch it before going into the ***running*** state. We can write a script where we launch an EC2 instance, wait until it reaches the running state, then we start to do something else

ec2-instance-running.sh

```
#!/bin/bash
instance_id=$(aws ec2 run-instances --image-id ami-12345 \
        --query Reservations[].Instances[].InstanceId \
        --output text)
instance_state=$(aws ec2 describe-instances --instance-ids $instance_id \
        --query 'Reservations[].Instances[].State.Name')
while [ "$instance_state" != "running" ]
do
    sleep 1
    instance_state=$(aws ec2 describe-instances --instance-ids $instance_id \
        --query 'Reservations[].Instances[].State.Name')
done
```

This is what such a code might look like.

ec2-instance-running.sh

```
#!/bin/bash
instance_id=$(aws ec2 run-instances --image-id ami-12345 \
        --query Reservations[].Instances[].InstanceId \
        --output text)
instance_state=$(aws ec2 describe-instances --instance-ids $instance_id \
        --query 'Reservations[].Instances[].State.Name')
while [ "$instance_state" != "running" ]
do
    sleep 1
    instance_state=$(aws ec2 describe-instances --instance-ids $instance_id \
        --query 'Reservations[].Instances[].State.Name')
done
```

We call the ***instance_id=$(aws ec2 run-instances - -image-id ami-12345 - -query Reservations[].Instances[].InstanceId - -output text)*** and save the created EC2 instance's ***InstanceId*** into a variable as above

ec2-instance-running.sh

```
#!/bin/bash
instance_id=$(aws ec2 run-instances --image-id ami-12345 \
        --query Reservations[].Instances[].InstanceId \
        --output text)
instance_state=$(aws ec2 describe-instances --instance-ids $instance_id \
        --query 'Reservations[].Instances[].State.Name')
while [ "$instance_state" != "running" ]
do
    sleep 1
    instance_state=$(aws ec2 describe-instances --instance-ids $instance_id \
        --query 'Reservations[].Instances[].State.Name')
done
```

We then get the state of the created instance and save it into the ***instance_state*** variable using the command ***instance_state=$(aws ec2 describe-instances - -instance-ids $instance_id - -query 'Reservations[].Instances[].State.Name')*** as above

ec2-instance-running.sh

```
#!/bin/bash
instance_id=$(aws ec2 run-instances --image-id ami-12345 \
        --query Reservations[].Instances[].InstanceId \
        --output text)
instance_state=$(aws ec2 describe-instances --instance-ids $instance_id \
        --query 'Reservations[].Instances[].State.Name')
while [ "$instance_state" != "running" ]
do
    sleep 1
    instance_state=$(aws ec2 describe-instances --instance-ids $instance_id \
        --query 'Reservations[].Instances[].State.Name')
done
```

Finally, we use a while loop command to poll until the EC2 instance is in the ***running*** state.

ec2-instance-running.sh

```
#!/bin/bash
instance_id=$(aws ec2 run-instances --image-id ami-12345 \
        --query Reservations[].Instances[].InstanceId \
        --output text)
instance_state=$(aws ec2 describe-instances --instance-ids $instance_id \
        --query 'Reservations[].Instances[].State.Name')
while [ "$instance_state" != "running" ]
do
    sleep 1
    instance_state=$(aws ec2 describe-instances --instance-ids $instance_id \
        --query 'Reservations[].Instances[].State.Name')
done
```

- No timeouts
- Failure states
- Hand-written code

There are bad issues with this code as above to prevent an infinite loop

ec2-waiters.sh

```
instance_id=$(aws ec2 run-instances --image-id ami-12345 \
        --query Reservations[].Instances[].InstanceId \
        --output text)
aws ec2 wait instance-running --instance-ids $instance_id
```

You can use the newly added ***wait*** command in the CLI as seen above to make the code better.

ec2-waiters.sh

```
instance_id=$(aws ec2 run-instances --image-id ami-12345 \
        --query Reservations[].Instances[].InstanceId \
        --output text)
aws ec2 wait instance-running --instance-ids $instance_id
```

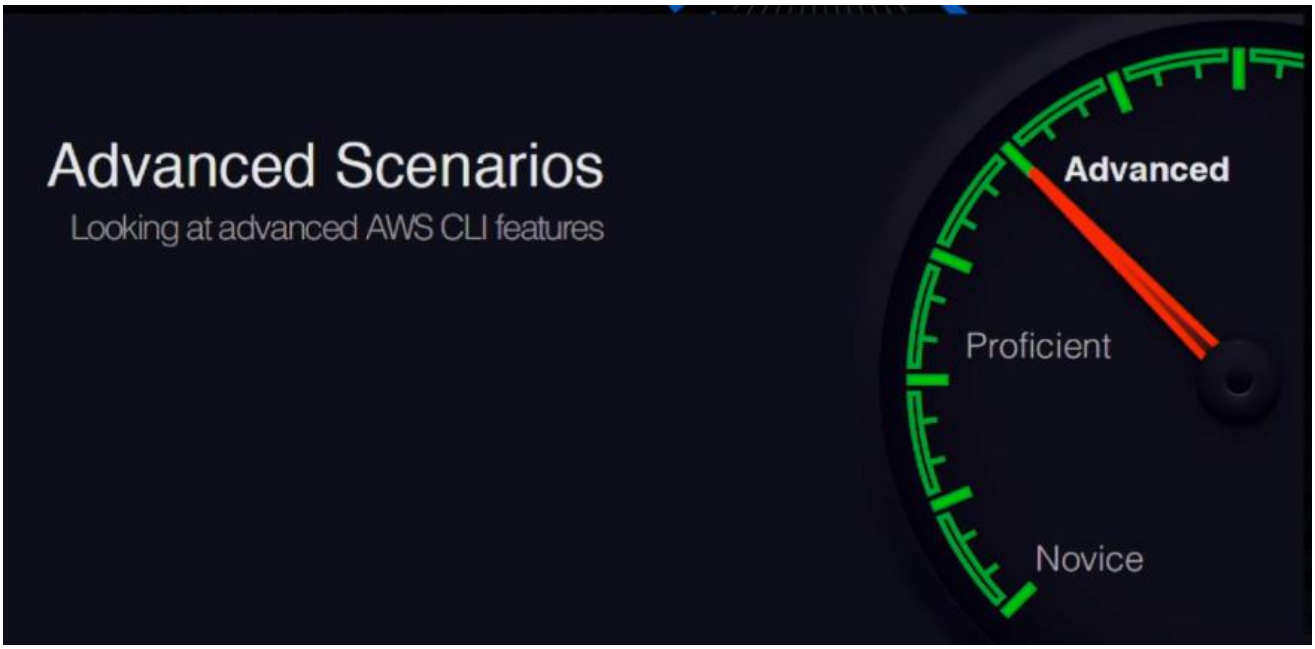
subcommand

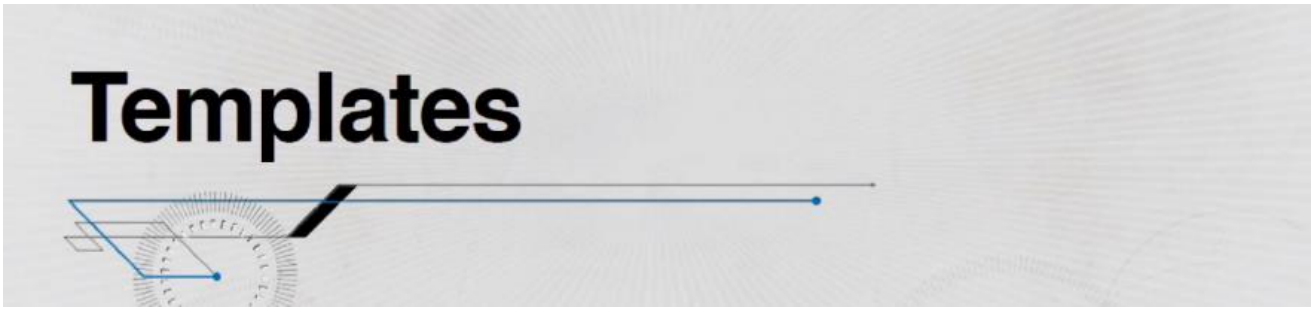
waiter name

describe-instances options

We use the ***wait*** command to block, this will block and will not exit until all of the instances are in a running state. The wait command corresponds to the describe-instances call. You can use the help command to get more information about all the waiters available in the CLI or run ***$ aws ec2 wait help*** or ***$ aws ec2 wait instance-running help*** commands.

If any of the instances were to terminate, the wait command will give you an error response telling you it couldn't wait for the terminated instance.

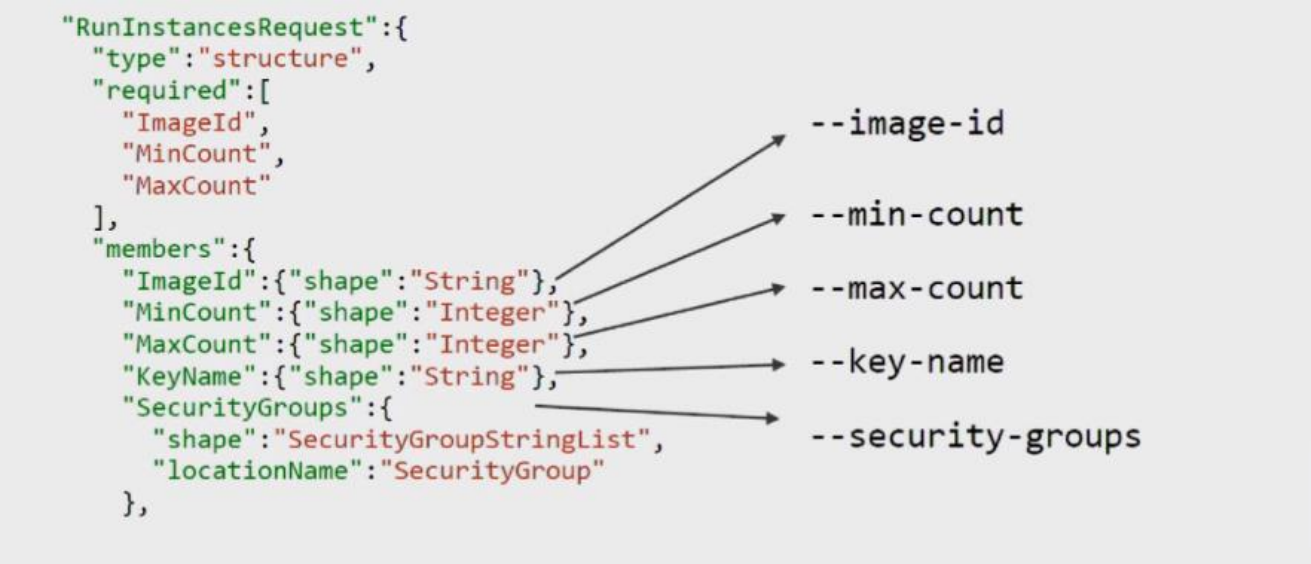




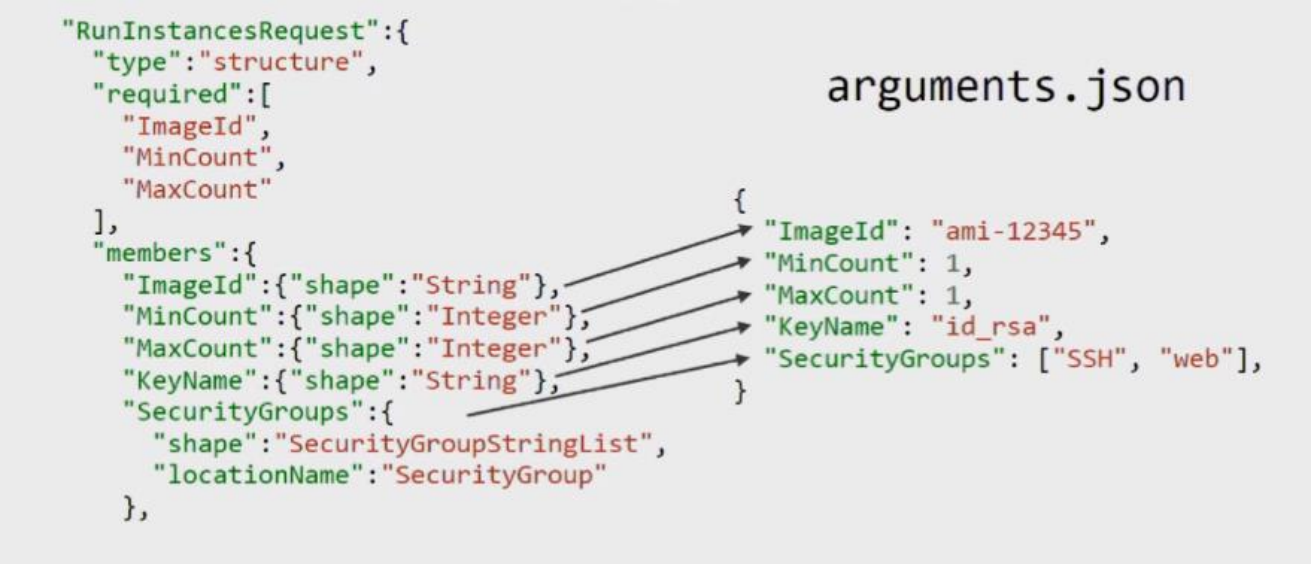
This means that there is a JSON model that describes all the inputs and all the outputs

```
"RunInstancesRequest":{
  "type":"structure",
  "required":[
    "ImageId",
    "MinCount",
    "MaxCount"
  ],
  "members":{
    "ImageId":{"shape":"String"},
    "MinCount":{"shape":"Integer"},
    "MaxCount":{"shape":"Integer"},
    "KeyName":{"shape":"String"},
    "SecurityGroups":{
      "shape":"SecurityGroupStringList",
      "locationName":"SecurityGroup"
    },
```

Somewhere in this model, we can introspect it at runtime map those to top level parameters that will surface in the CLI



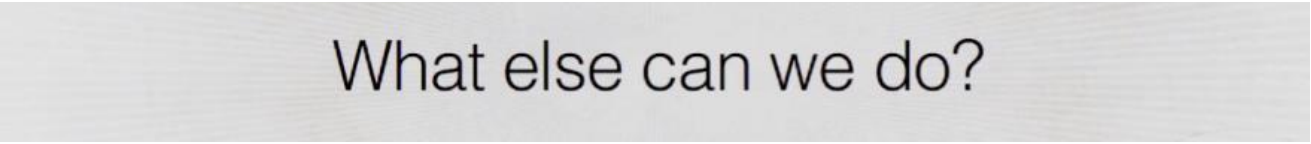
This makes it possible for us to accept alternate input formats



Like we can just accept all the arguments to a command from a single JSON file as above, instead of having all the arguments listed out with the command using flags.

```
aws ec2 run-instances --cli-input-json file://arguments.json
```

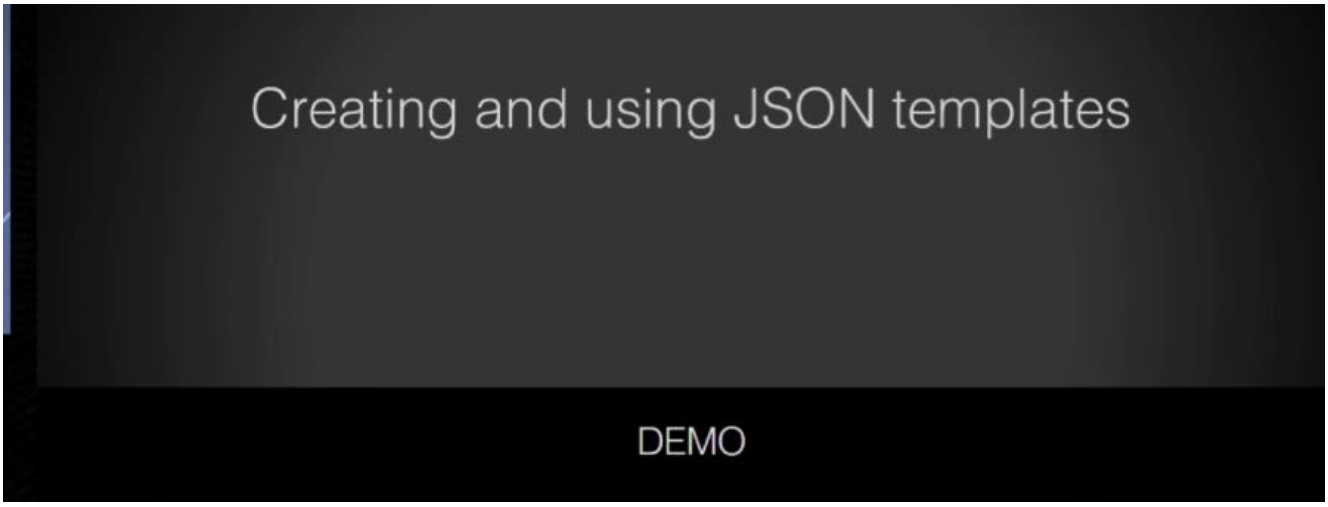
You can now run your CLI commands and pass the arguments from a file using the ***- -cli-input-json*** flag and passing the relative file path as above.

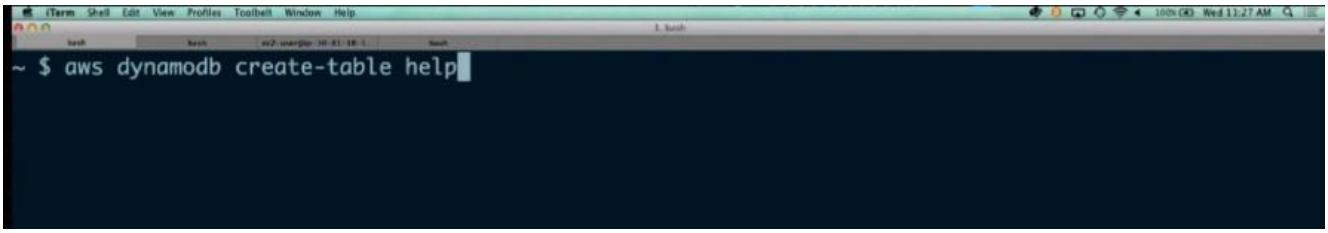


Because we have an input model that tells us exactly how the input structure needs to look like, we can also generate what that sample input looks like

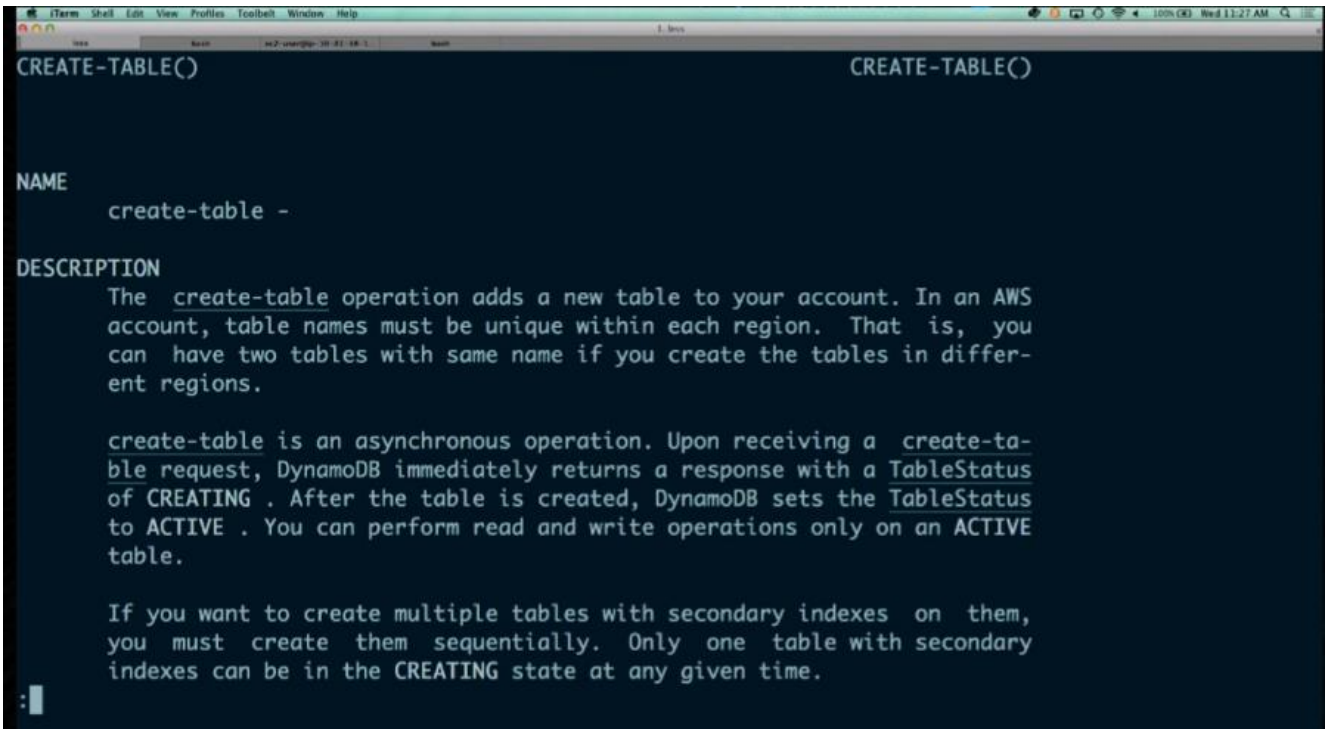
```
aws ec2 run-instances --generate-cli-skeleton
```

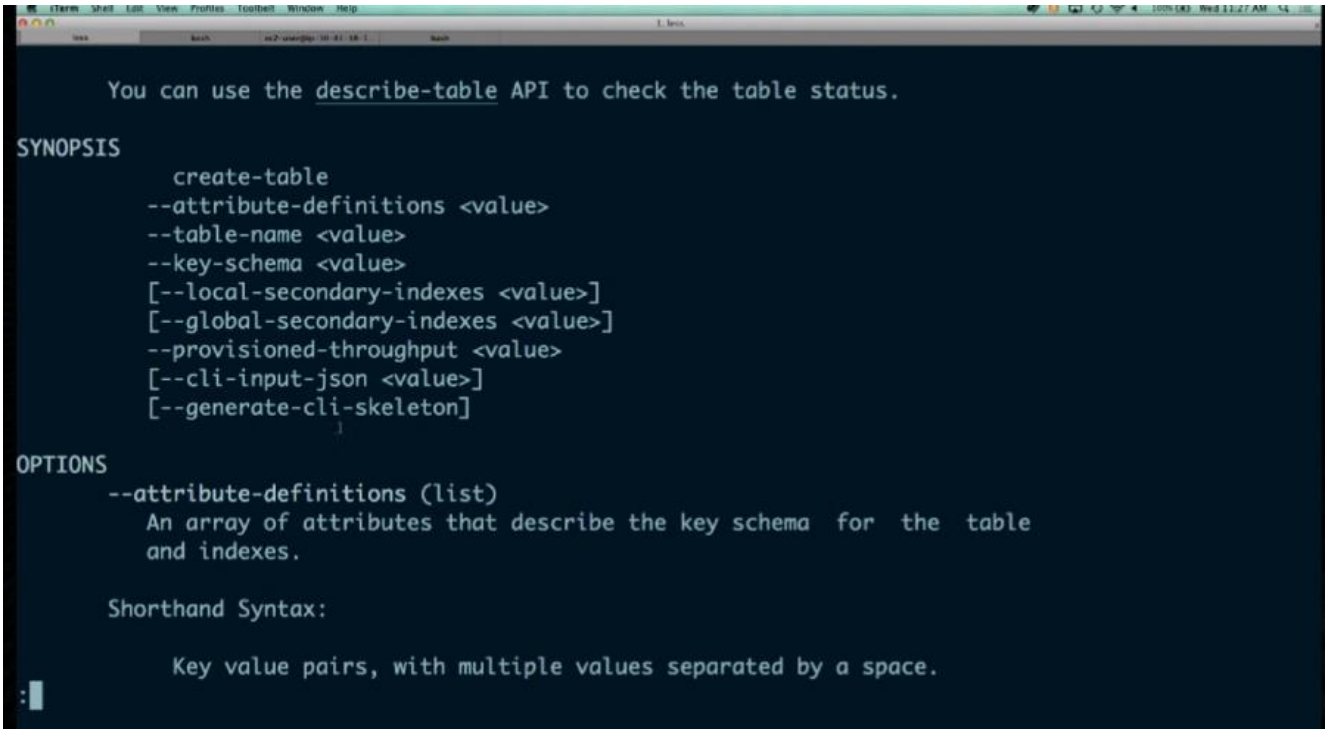
You can generate what the input of a command should look like by using the ***- -generate-cli-skeleton*** command. This will generate a sample JSON of what the input should look like and write it out to stdout.



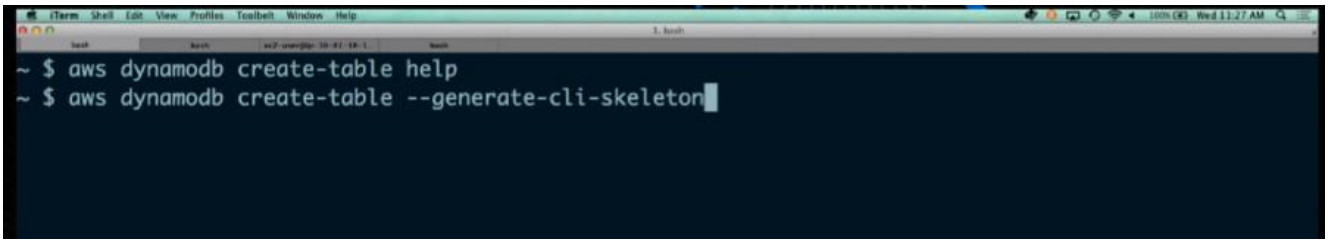


We can use the ***$ aws dynamodb create-table help*** command

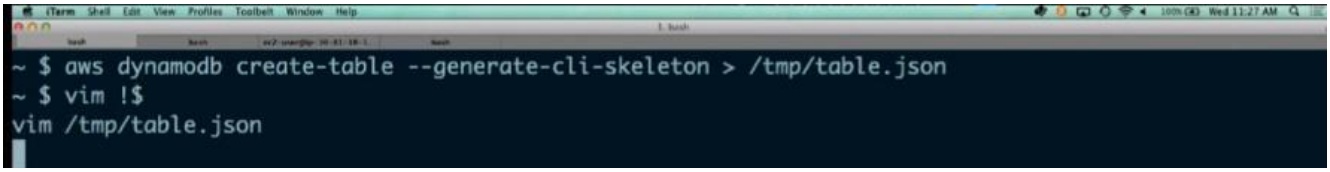




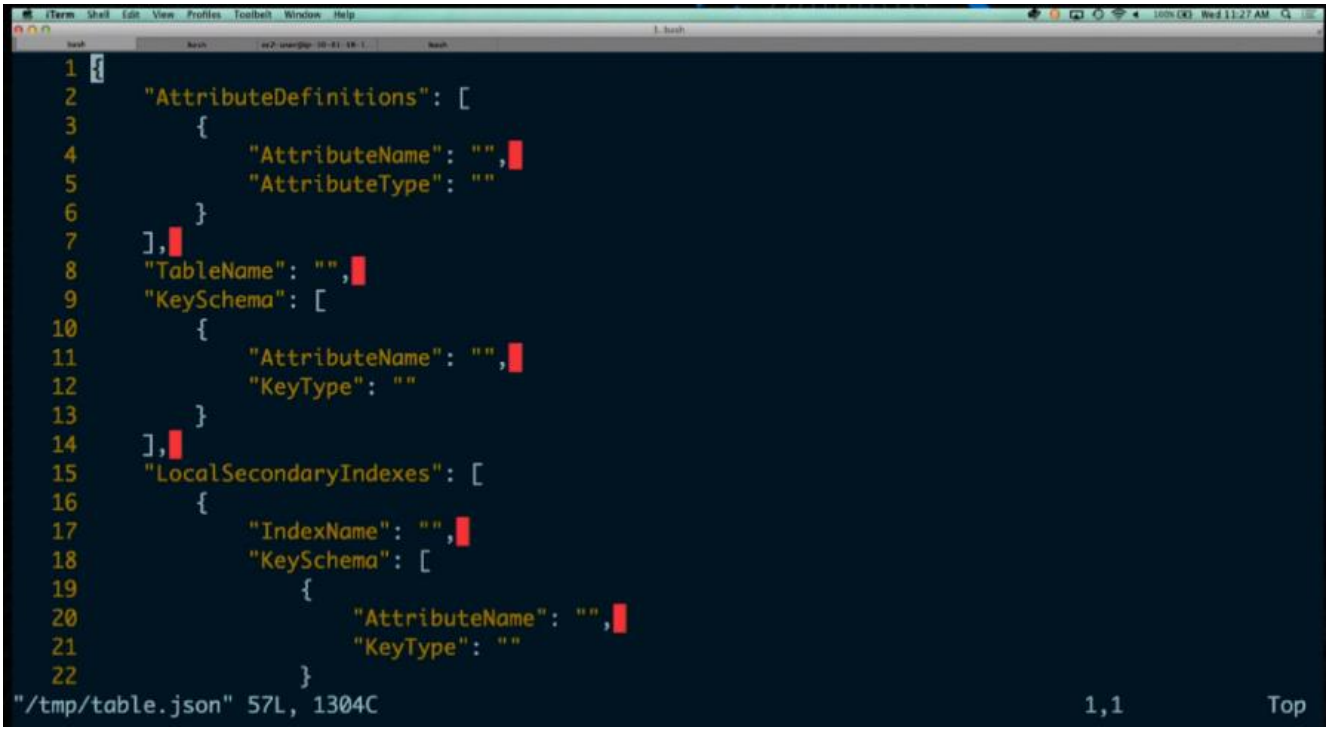
We can also see what a sample syntax looks like as above



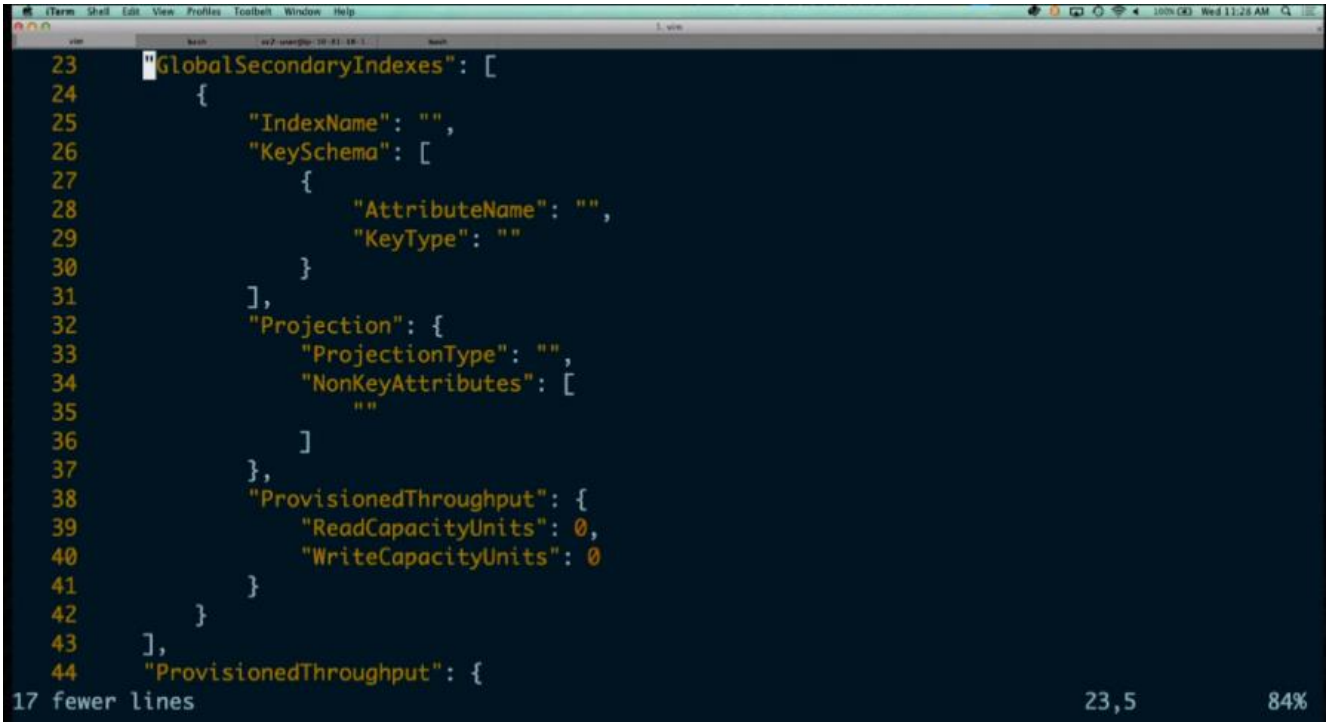
We can instead use the ***$ aws dynamodb create-table - -generate-cli-skeleton*** command to generate a sample JSON input for the create-table command we are trying to run



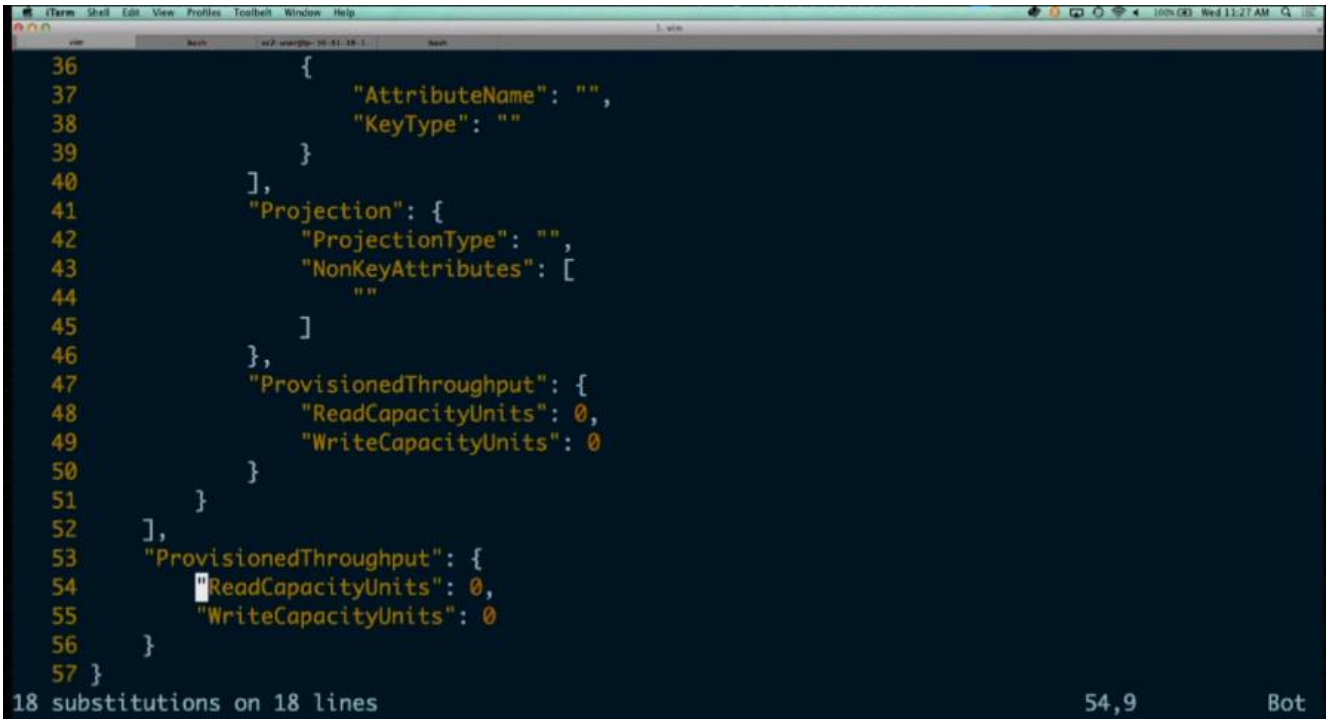
We can save the sample input JSON to a file using the ***$aws dynamodb create-table - -generate-cli-skeleton > /tmp/table.json*** command as above. Then we can open up that file to see what the sample input format is like



Notice that it has the structure of the needed inputs, we can now go in and fill in the values we want to use for the input and delete options that we don't want



```
23      "GlobalSecondaryIndexes": [
24          {
25              "IndexName": "",
26              "KeySchema": [
27                  {
28                      "AttributeName": "",
29                      "KeyType": ""
30                  }
31              ],
32              "Projection": {
33                  "ProjectionType": "",
34                  "NonKeyAttributes": [
35                      ""
36                  ]
37              },
38              "ProvisionedThroughput": {
39                  "ReadCapacityUnits": 0,
40                  "WriteCapacityUnits": 0
41              }
42          }
43      ],
44      "ProvisionedThroughput": {
17 fewer lines                                          23,5          84%
```



```
36                  {
37                      "AttributeName": "",
38                      "KeyType": ""
39                  }
40              ],
41              "Projection": {
42                  "ProjectionType": "",
43                  "NonKeyAttributes": [
44                      ""
45                  ]
46              },
47              "ProvisionedThroughput": {
48                  "ReadCapacityUnits": 0,
49                  "WriteCapacityUnits": 0
50              }
51          }
52      ],
53      "ProvisionedThroughput": {
54          "ReadCapacityUnits": 0,
55          "WriteCapacityUnits": 0
56      }
57  }
18 substitutions on 18 lines                            54,9          Bot
```

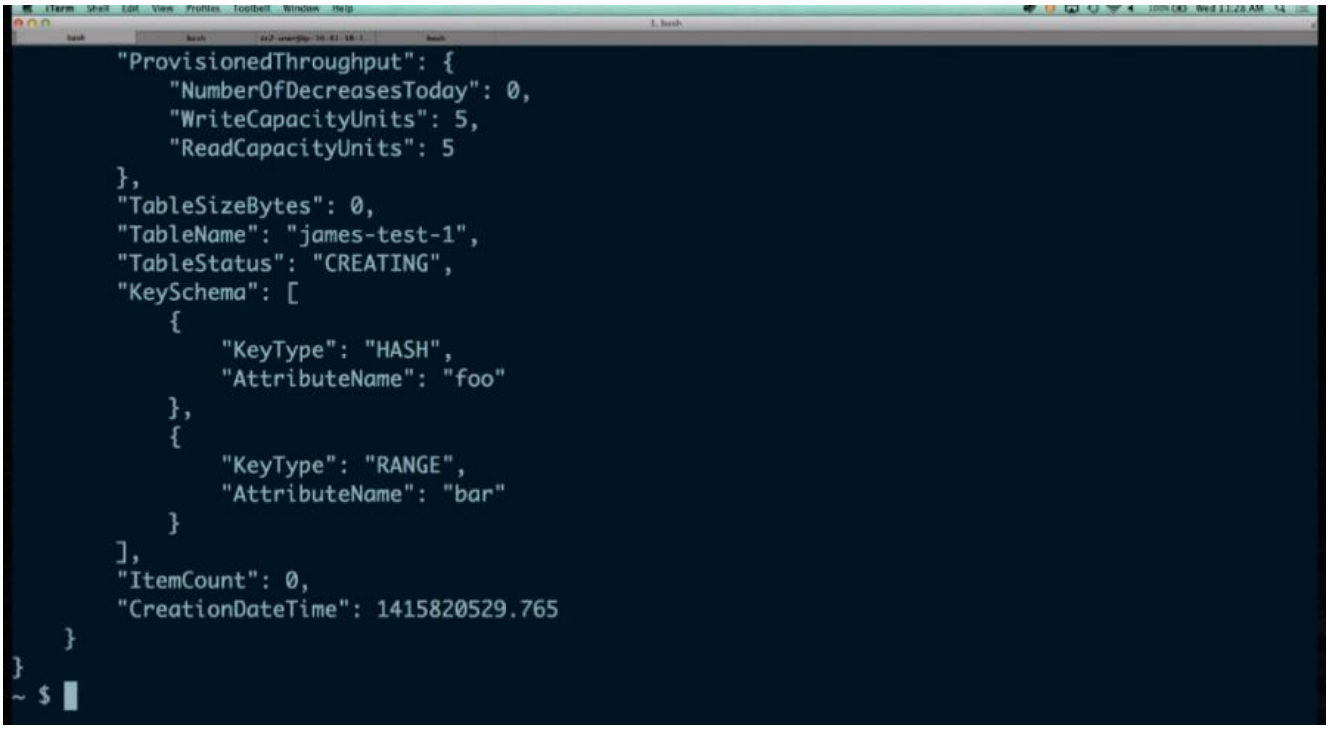
```
{
    "AttributeDefinitions": [
        {
            "AttributeName": "foo",
            "AttributeType": "S"
        },
        {
            "AttributeName": "bar",
            "AttributeType": "N"
        }
    ],
    "TableName": "james-test-1",
    "KeySchema": [
        {
            "AttributeName": "foo",
            "KeyType": "HASH"
        },
        {
            "AttributeName": "bar",
            "KeyType": "RANGE"
        }
    ],
-- INSERT --                                              20,30         Top
```

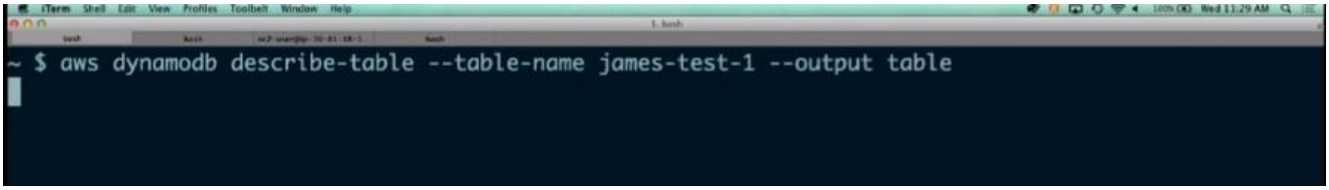
We can now start filling in the values we want to use as inputs to the create-table command

```
    "ProvisionedThroughput": {
        "ReadCapacityUnits": 5,
        "WriteCapacityUnits": 5
    }
}
~
~
~
```

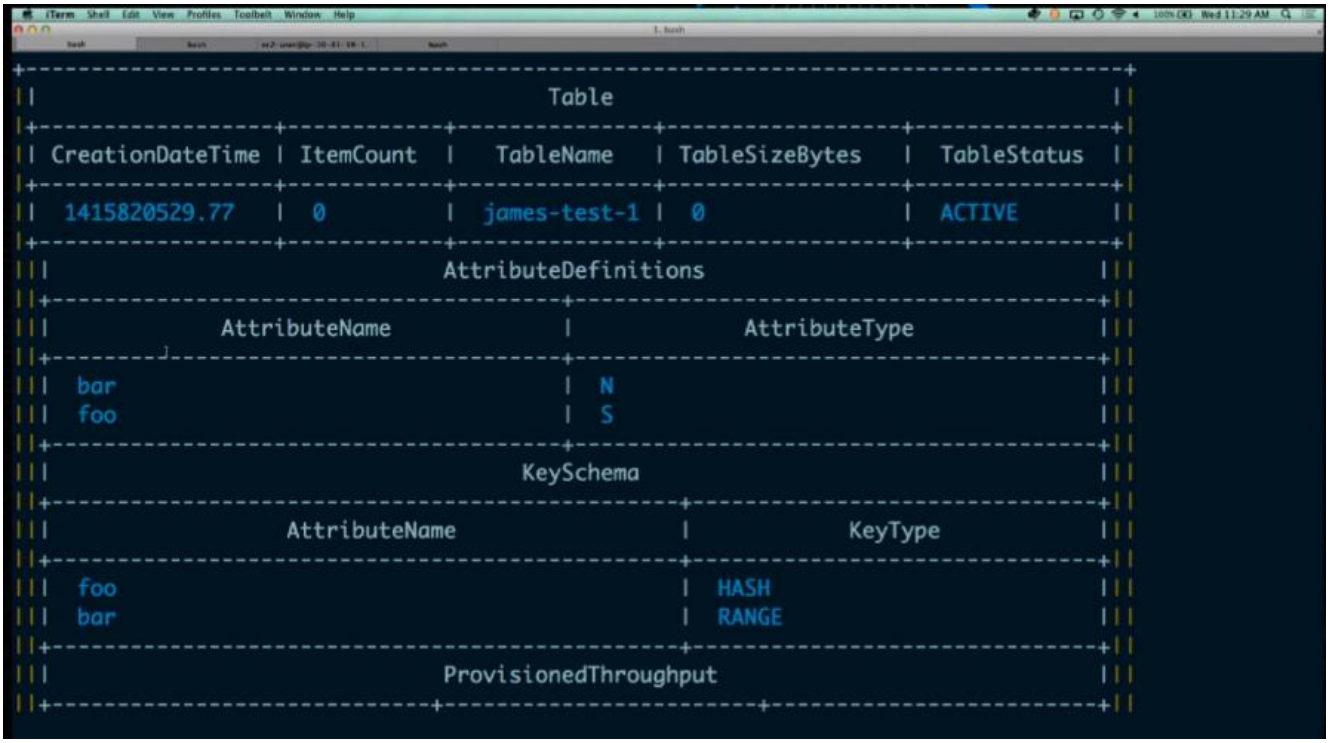
```
~ $ aws dynamodb create-table --generate-cli-skeleton > /tmp/table.json
~ $ vim !$
vim /tmp/table.json
~ $ aws dynamodb create-table --cli-input-json file:///tmp/table.json
```

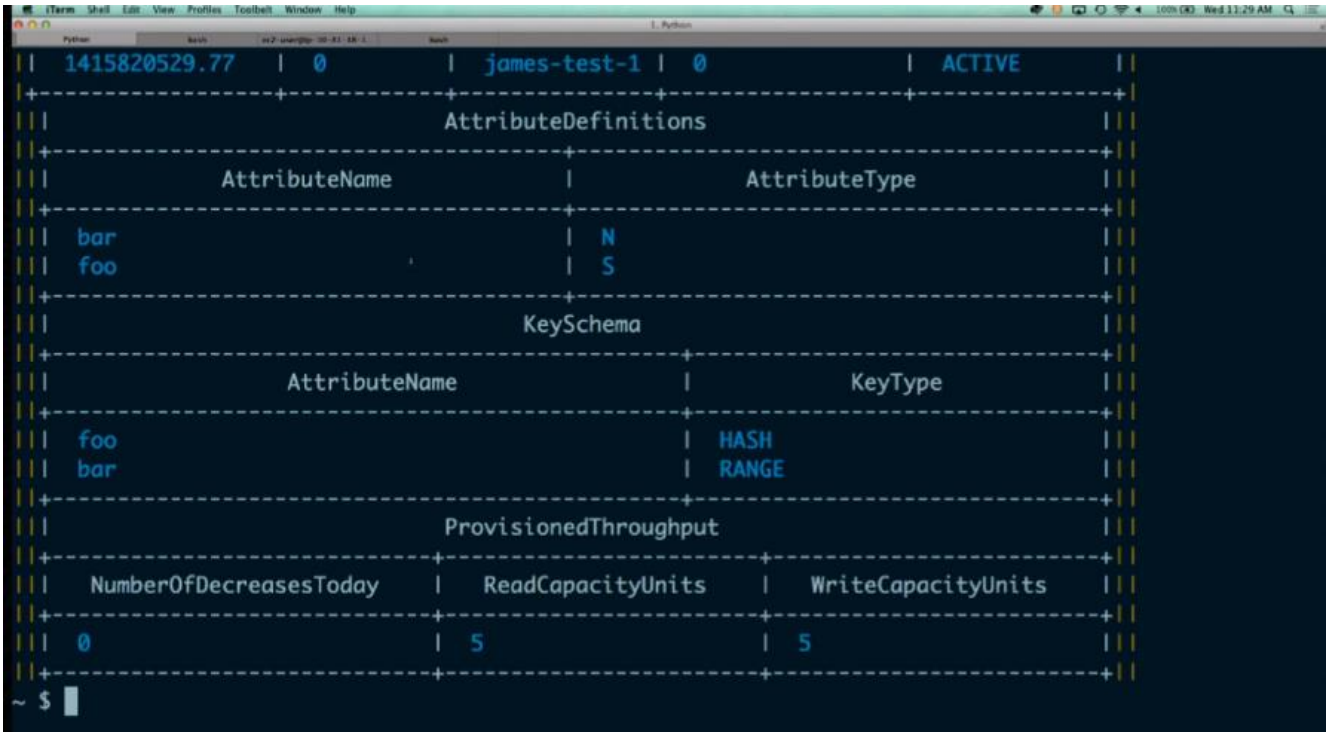
We can now issue the CLI command and pass the file as an input file using ***$ aws create-table - -cli-input-json file:///tmp/table.json*** to have the DynamoDB table created for us



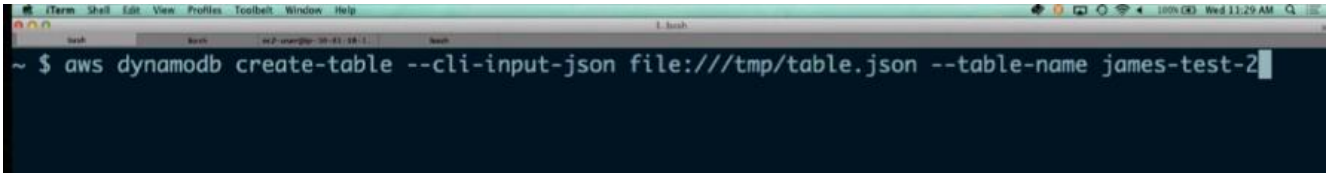


We can now use the ***$ aws dynamodb describe-table - -table-name james-test-1 - -output table*** command to see the details of the created table using the table display format as below

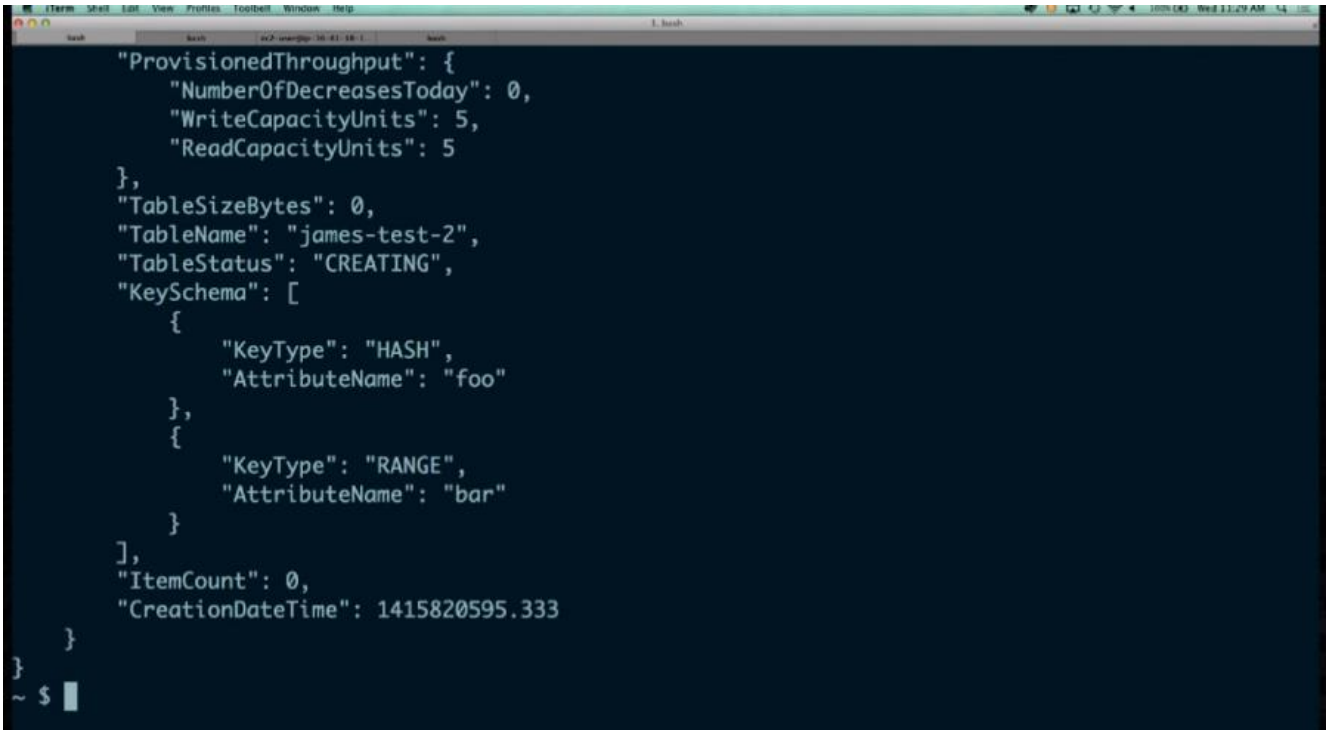




It has all the parameters that we specified in the input JSON document



We can also specify overwrites to the input details in the input JSON file by passing the overwrites as flags within the command as in ***$ aws dynamodb create-table - -cli-input-json file:///tmp/table.json - -table-name james-test-2*** above



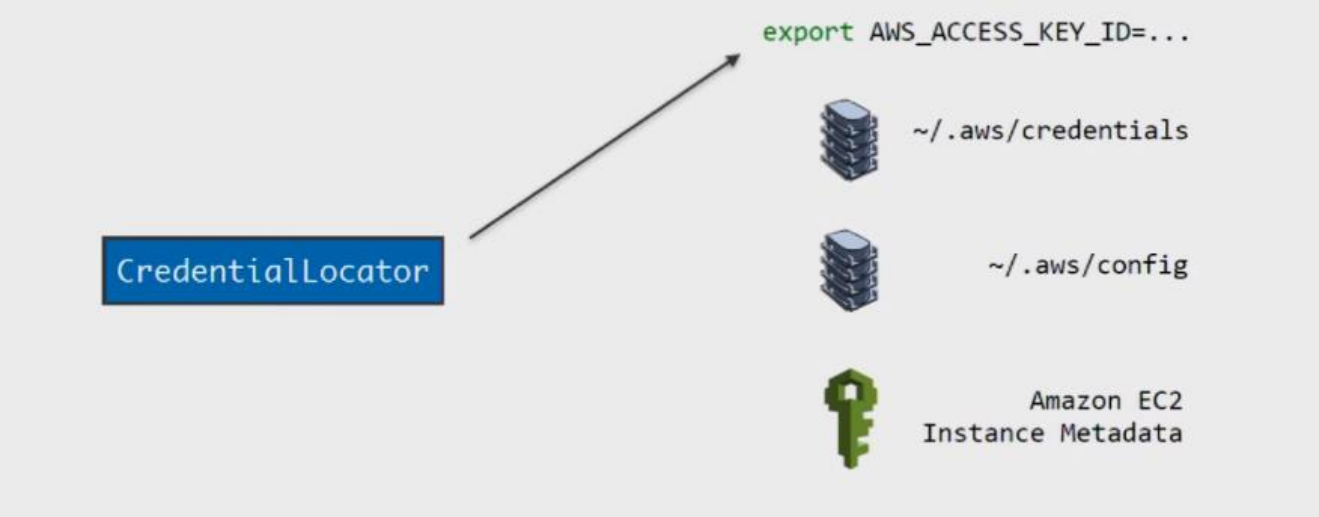
```
~ $ aws dynamodb describe-table --table-name james-test-2 --output table
```

```
|                                  DescribeTable                                  |
+---------------------------------------------------------------------------------+
||                                    Table                                      ||
|+------------------+------------+---------------+------------------+-----------+|
|| CreationDateTime | ItemCount  |   TableName   | TableSizeBytes   | TableStatus||
|+------------------+------------+---------------+------------------+-----------+|
||  1415820595.33   |  0         |  james-test-2 |  0               |  CREATING  ||
|+------------------+------------+---------------+------------------+-----------+|
|||                          AttributeDefinitions                               |||
||+-------------------------------------+-------------------------------------+||
|||         AttributeName               |            AttributeType            |||
||+-------------------------------------+-------------------------------------+||
|||  bar                                |  N                                  |||
|||  foo                                |  S                                  |||
||+-------------------------------------+-------------------------------------+||
|||                                  KeySchema                                  |||
||+---------------------------------------------+-----------------------------+||
|||              AttributeName                  |           KeyType           |||
||+---------------------------------------------+-----------------------------+||
|||  foo                                        |  HASH                       |||
|||  bar                                        |  RANGE                      |||
||+---------------------------------------------+-----------------------------+||
```

We now have the table created with the overwrite table name

This refers to the mechanism in which the CLI looks up credentials to use for requests to signed with.

This is the credentials lookup process that the CLI uses.



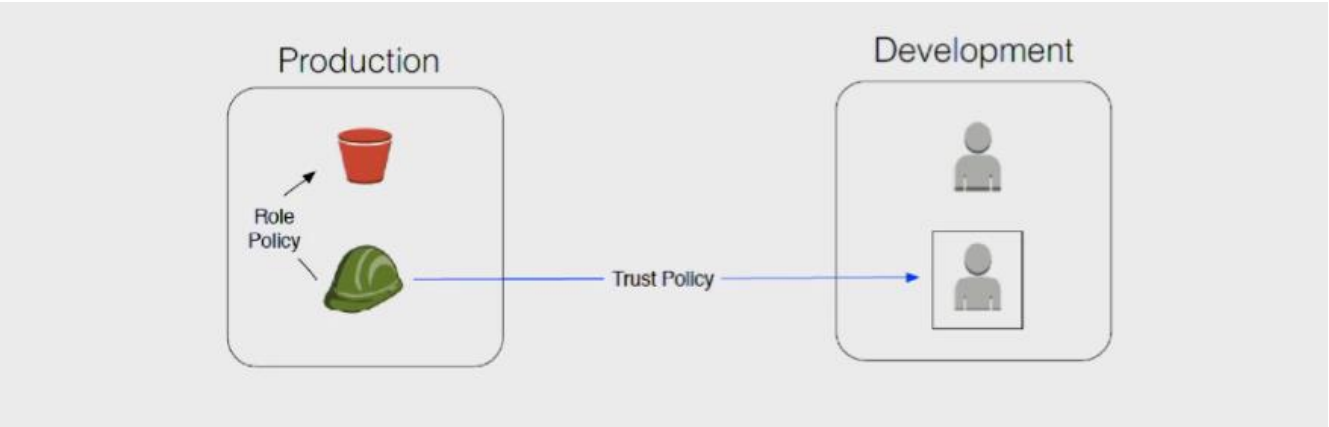
First, it asks the environment credentials provider if it has any credentials stored in it. The answer to this is NO

Then it asks the ~/.aws/credentials file for credentials, it get the credentials to use for the duration of the process. Note that ***it is possible to have new credentials providers plugged into this lookup process***

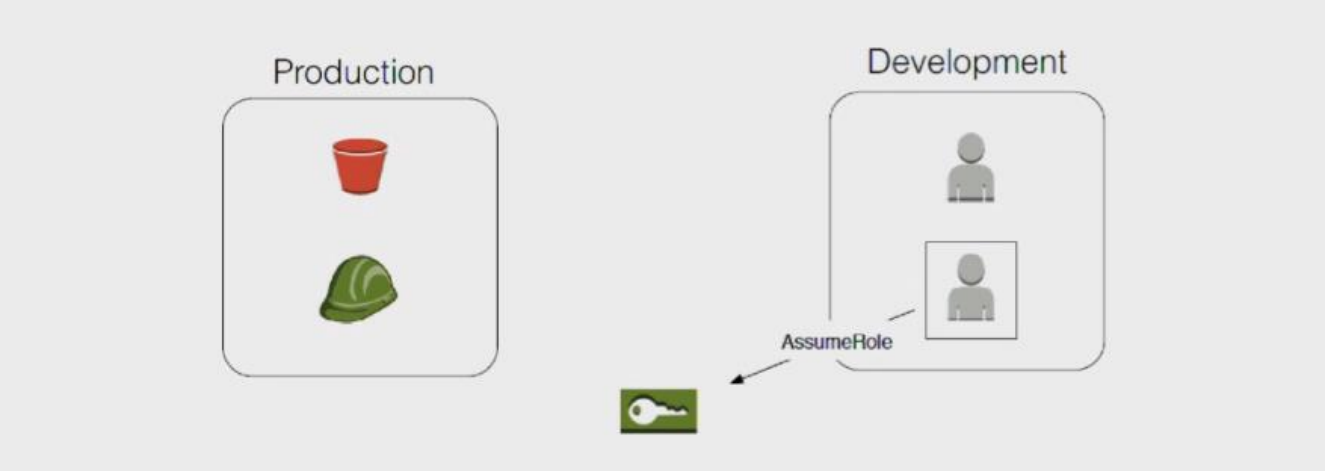
The ***AssumeRole*** provider is a newly added ***credentials provider*** that was just added to the lookup process,

Let us look at the example where we want cross-account access using IAM roles, we sill see how the new AssumeRole credentials provider helps us achieve this.

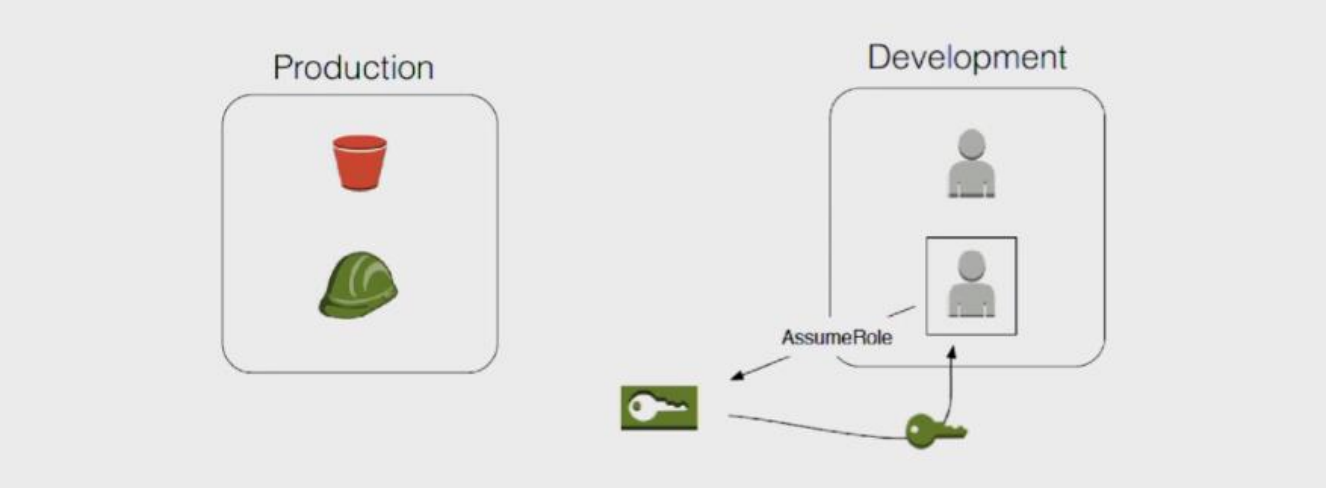
Assume we have 2 separate AWS accounts, one that has an S3 bucket in Production that we would like to have access to from the other Development account that has 2 IAM users. We want to give the IAM user on the bottom access to that S3 bucket. We can create an IAM role (an IAM role just has 2 components, we need to say who can assume the role, and what permissions do they have after assuming that role).



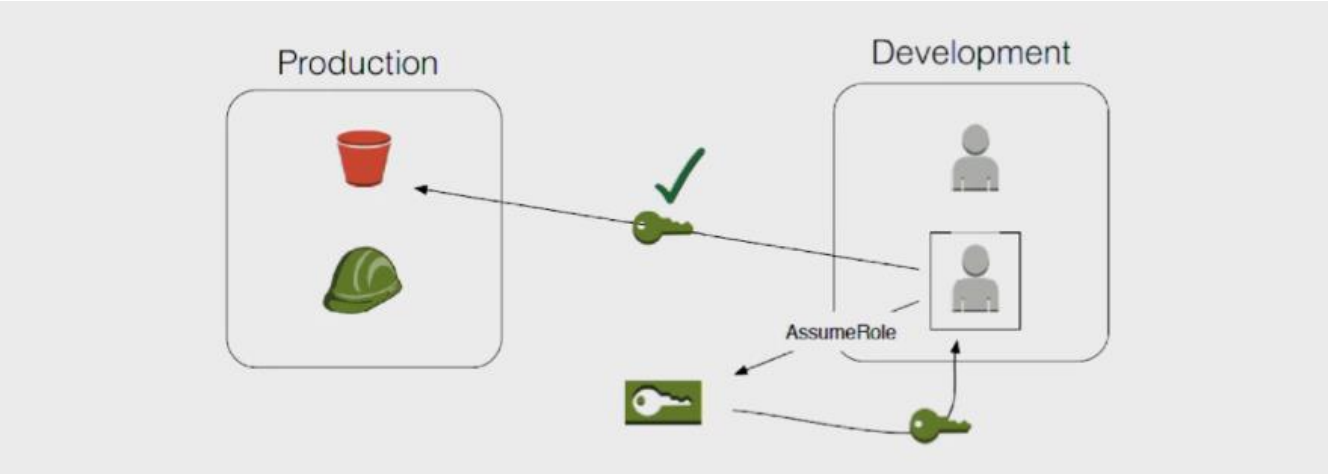
We are going to create a Role and say that the bottom developer/IAM user has access to the S3 bucket once it has assumed the role we created.



You will need to make a call to the AWS Security Token Service ***STS*** saying you want to assume the role, STS will then check to make sure that you have access to assume that role,



If you indeed are allowed to assume that role, STS will give you a set of temporary credentials.

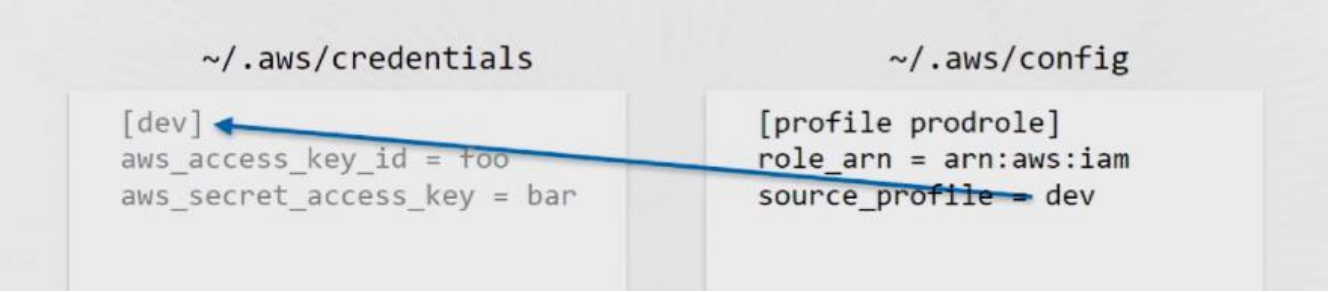


Those temporary credentials can then be used to assume the role and do things you wanted to do. This are temporary credentials that will expire. The ***AssumeRole*** provider makes this flow easy to achieve by managing the process for you in the CLI. By using the 2 commands below

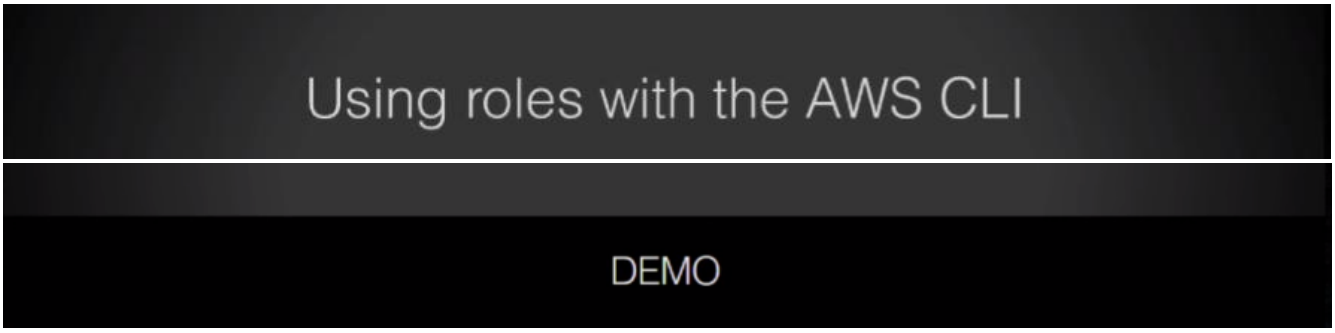
configure-role.sh

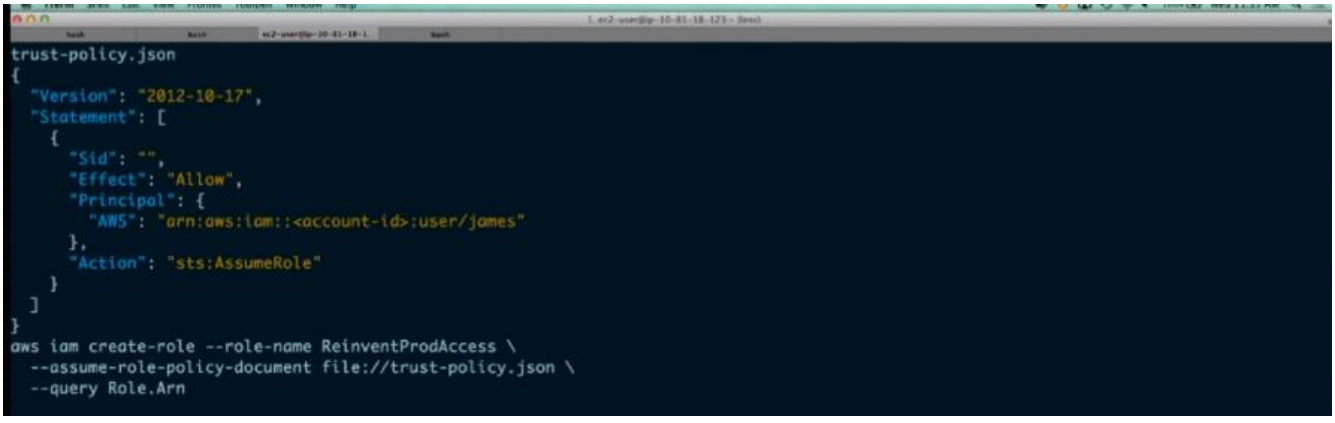
```
aws configure set profile.prodrole.role_arn arn:aws:iam...

aws configure set profile.prodrole.source_profile dev
```

We can use the ***AssumeRole provider*** with the 2 commands ***$ aws configure set profile.prodrole.role_arn arn:aws:iam…*** to specify the Role that we want to assume and the ***$ aws configure set profile.prodrole.source_profile dev*** to set the source profile.



The source profile is pointing to another profile that as credentials, this is because the initial assume_role call is an authenticated call. But we still need credentials. Notice that dev is pointing to existing profile somewhere.

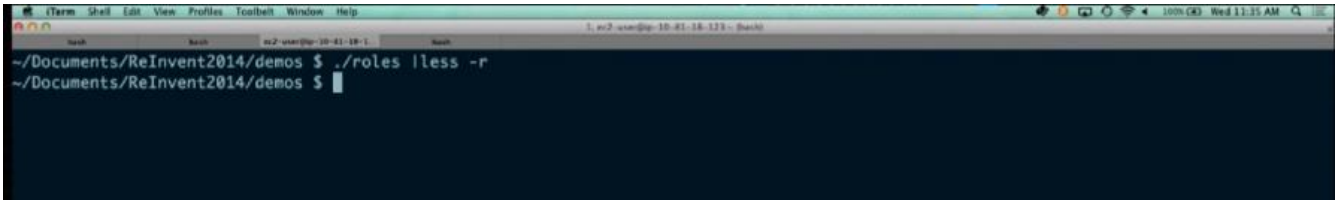




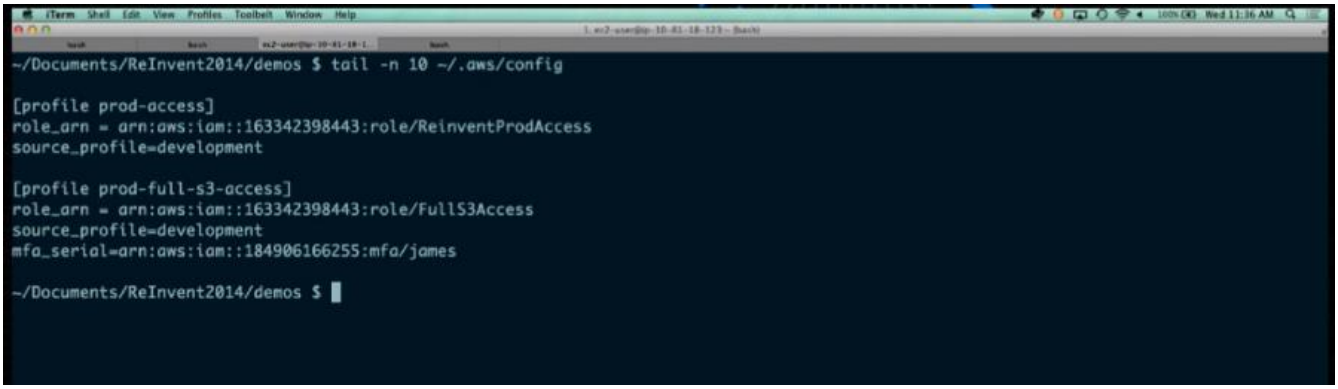
We have already set up a role shown above. We are going to create a role using the ***$ aws iam create-role - -role-name ReinventProdAccess - -assume-role-policy-document file://trust-policy.json - -query Role.Arn*** command and give it the ***trust-policy.json*** document that just says that a particular user ***james*** is allowed to assume the role.

```
role-policy.json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["s3:ListAllMyBuckets"],
      "Resource": "arn:aws:s3:::*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket",
        "s3:GetBucketLocation"
      ],
      "Resource": "arn:aws:s3:::jamesls-reinvent-role"
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:*"
      ],
      "Resource": "arn:aws:s3:::jamesls-reinvent-role/*"
    }
  ]
}
aws iam put-role-policy --role-name ReinventProdAccess \
    --policy-name S3BucketAccess --policy-document file://role-policy.json
(END)
```

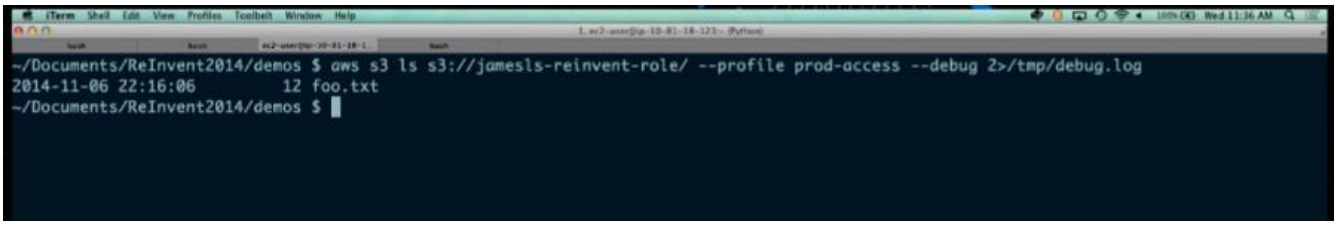
Then we are going to give it access to an S3 bucket using the ***$ aws iam put-role-policy - -role-name ReinventProdAccess - -policy-name S3BucketAccess - -policy-document file://role-policy.json*** command that says the user can List (ListBucket) and Get the bucket location (GetBucketLocation). Can do any operation within the S3 bucket ("s3:*"), and can also list all the buckets ("s3:ListAllMyBuckets").



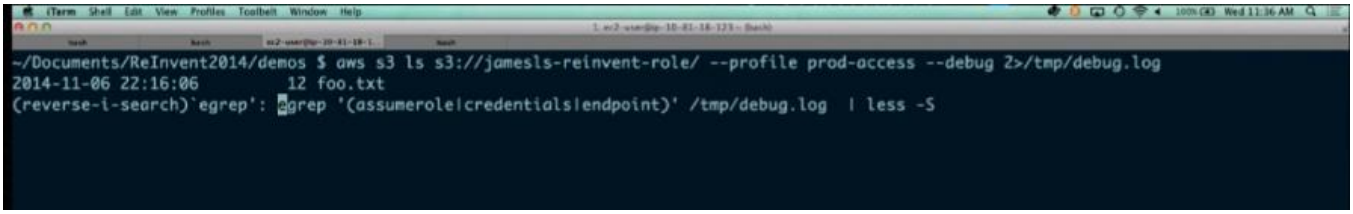
This now gives us the ability to assume the role once we run those 2 commands that configure set commands



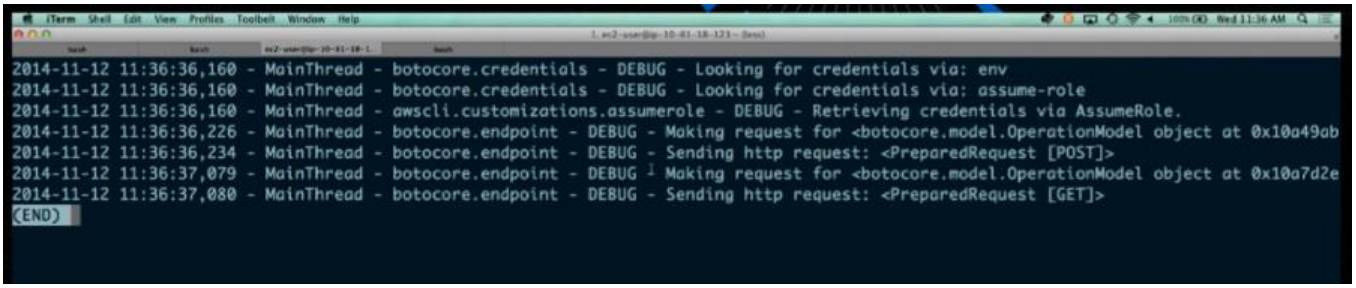
We can see the setting for the prod-access profile with the role_arn and the source_profile above.

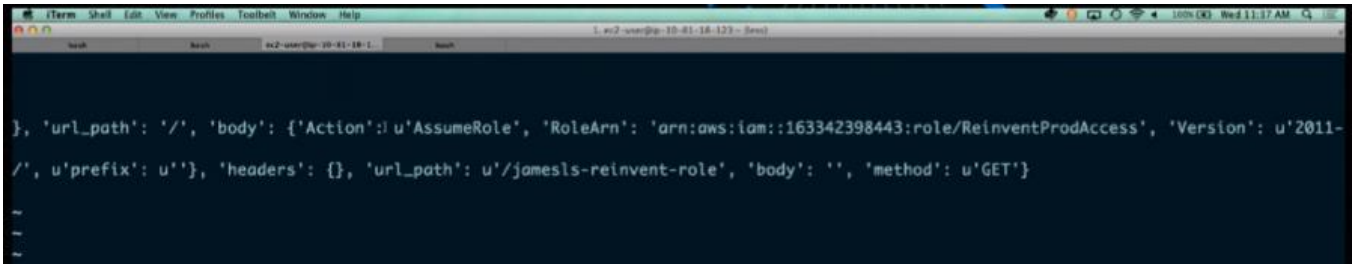


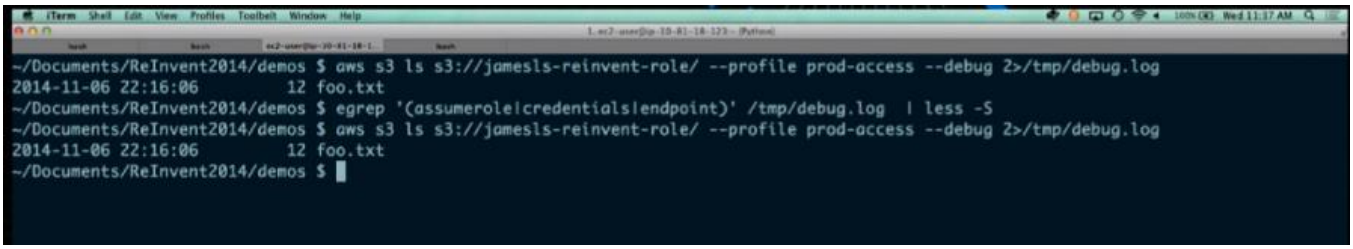
We then issue the ***$ aws s3 ls s3://jamesls-reinvent-role/ - -profile prod-access - -debug 2>/tmp/debug.log*** list files command and attach a debug log with the - -debug flag as above. It is then going to assume the role and then reuse those temporary credentials to make the S3 list call. Note that we see the output of the list of files in the S3 bucket.

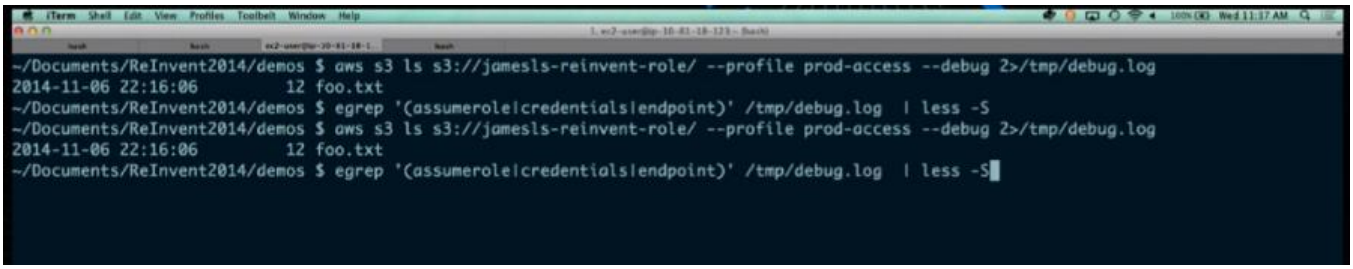


We can now check the debug log created for the command execution

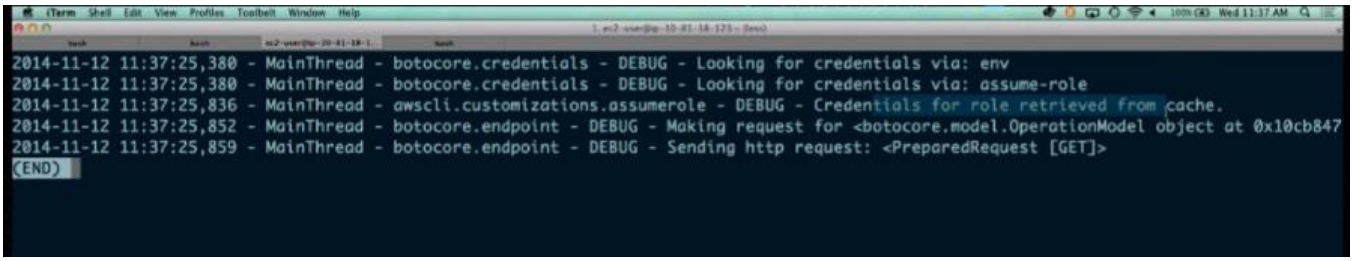


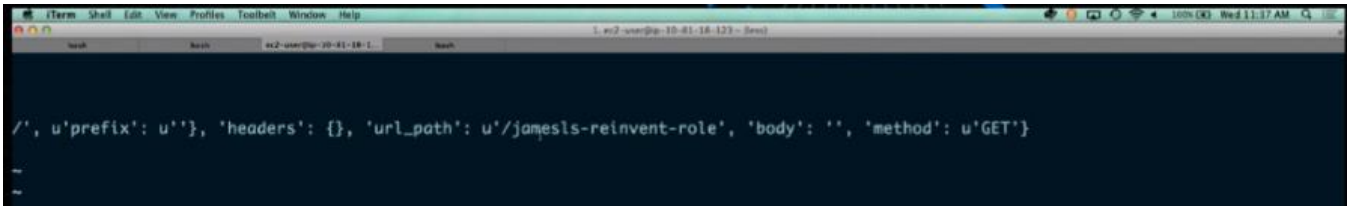


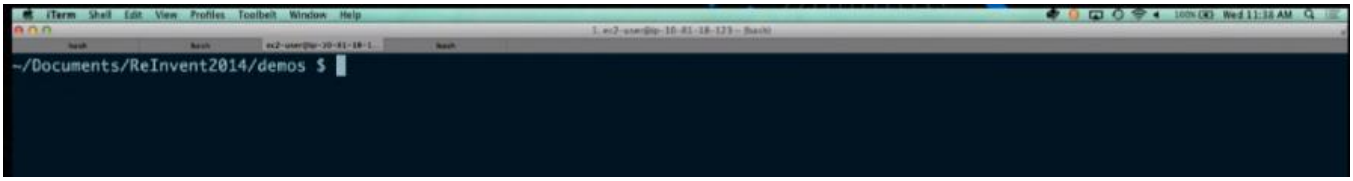




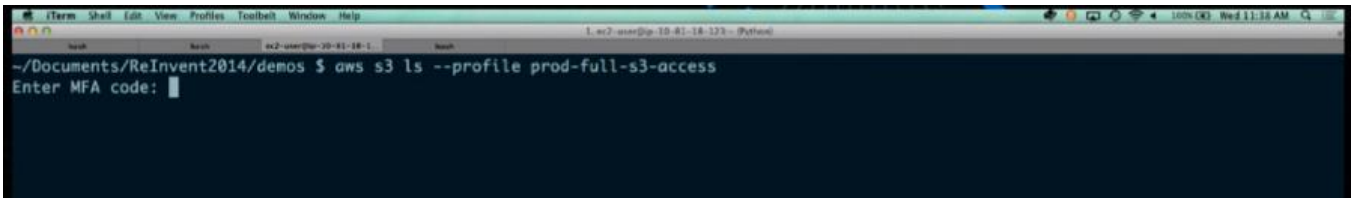
We can make the call again to show that it just reuses the assume role it now has to make the S3 call



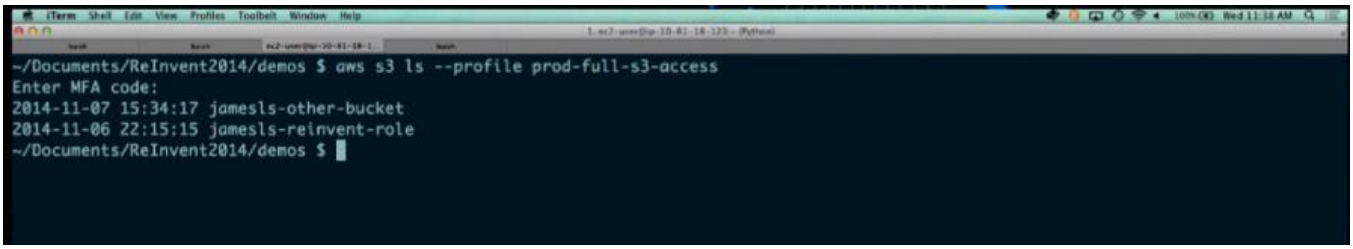


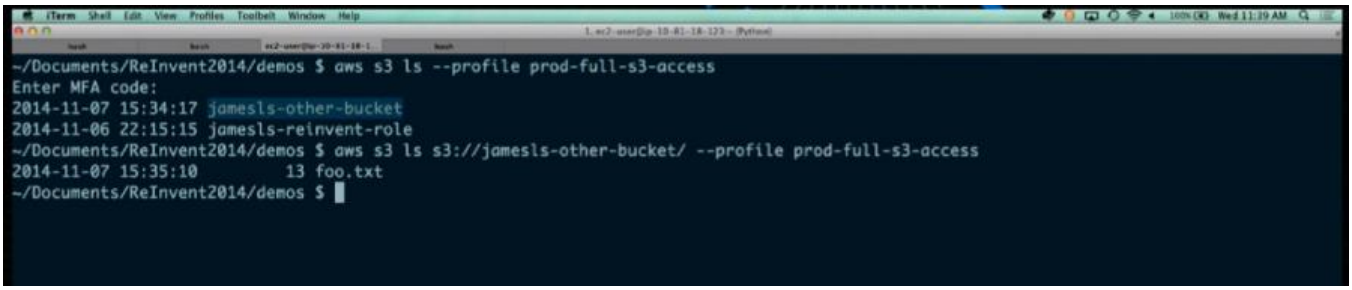


Note the profile prod-full-s3-access above, note that it also has a role_arn, source_profile, and an additional line for ***mfa_serial***. This is the ability that when assuming a role, you also want to require that they also send a one-time passcode from an ***MFA device*** before being allowed to do calls.



We then need to enter the MFA code before this call can get executed as below





We can also use the cached credentials to make more calls as above

```
aws s3 cp
```

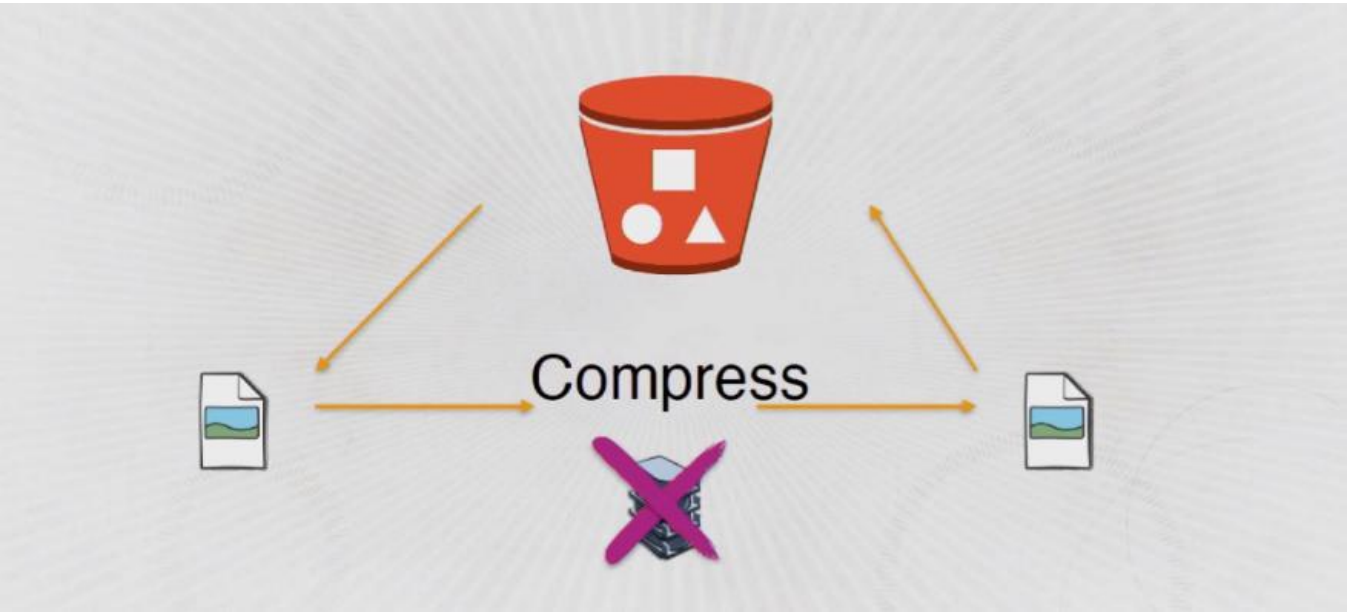
We want to avoid disk

```
aws s3 cp - s3://bucket/key
```

Instead of reading from a particular file, it will stream from stdin.

```
aws s3 cp s3://bucket/key -
```

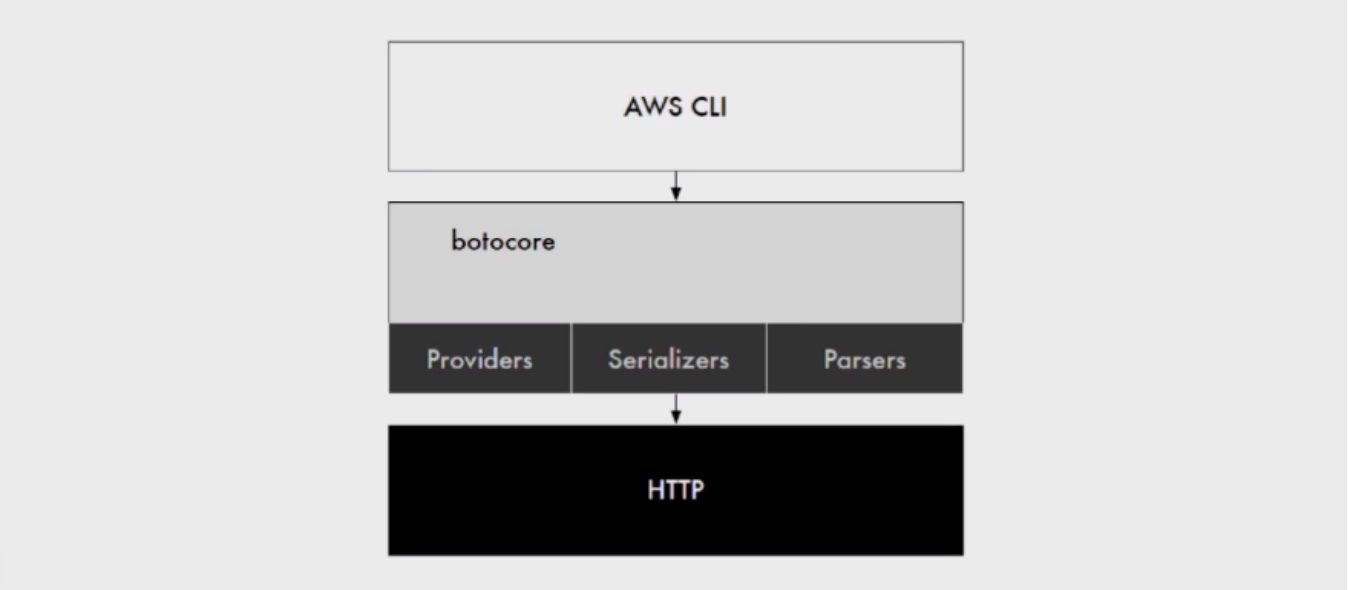
This also allows you to stream to stdout.



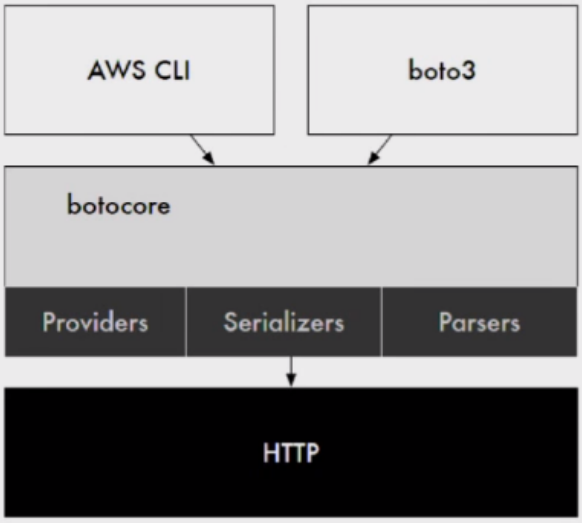
This is useful for the above use case where we want to take a file from S3 via download, compress it, and upload it back to S3.

```
aws s3 cp s3://bucket/key - | \
bzip2 --best | \
aws s3 cp - s3://bucket/key.bz2
```

We can instead use something like above where we are running the cp command, taking it and streaming it out to stdout, then piping it into a command called bzip2 to compress the data, we then take the result and stream it back up to S3. This code can handle file of any size where we don't want to write out temporary files.



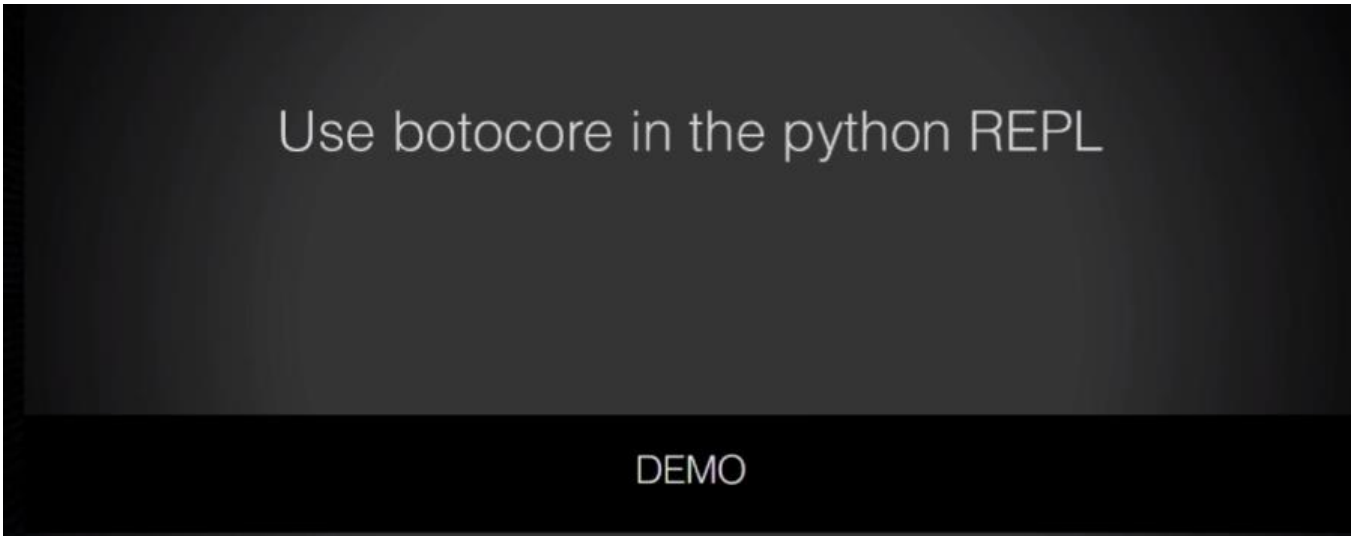
Botocore is the Python library that the CLI uses, it handles all the requests serialization, response parsing, credentials management, etc for the CLI. The CLI uses botocore as a building block library that other libraries like boto3 can use.

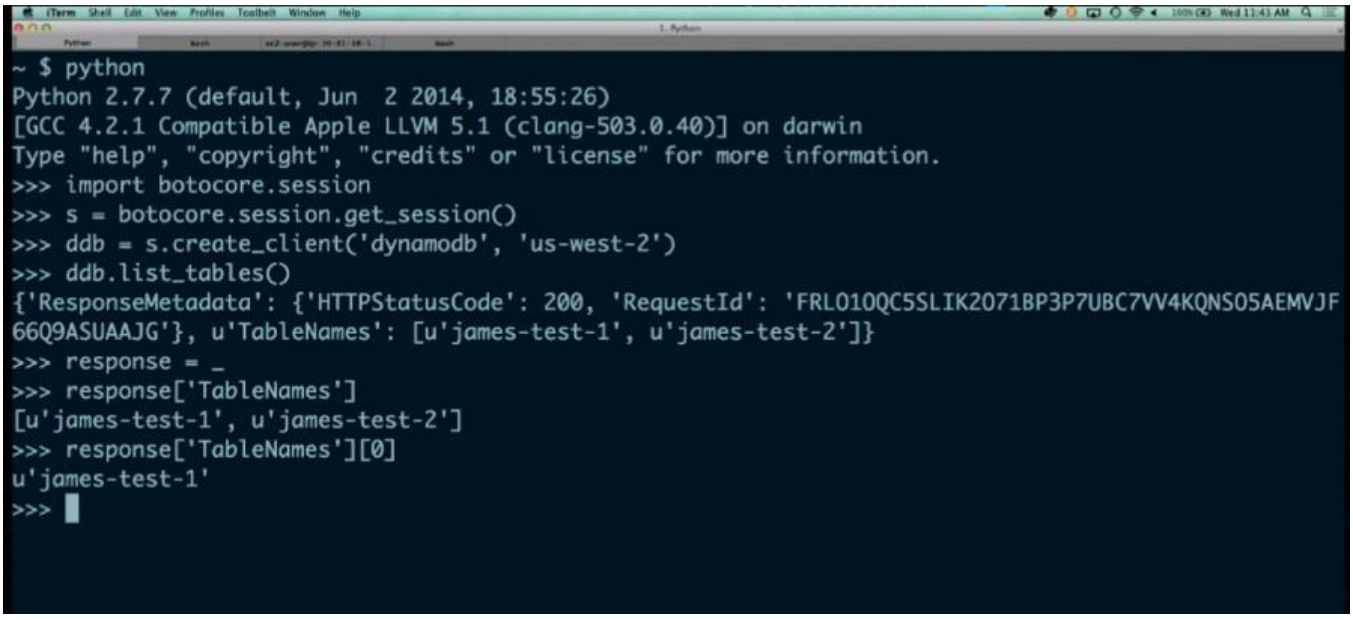


list_buckets.py

```
import botocore

s = botocore.session.get_session()
s3 = s.create_client('s3', region_name='us-west-2')

for bucket in s3.list_buckets()['Buckets']:
    print("Bucket: {bucket}".format(bucket=bucket['Name']))
```

If you installed the AWS CLI with pip, then you already have to the botocore library installed for you. You can then start using it with Python to explore all what you can use it to do with AWS. We create a session, then a low-level client for an AWS service like S3, then we get all buckets and print out the names





Wrapping Up

- Configuration
- Waiters
- Query
- Templates
- Credential Providers
- Amazon S3 Streaming

For More Information

- https://github.com/aws/aws-cli
- http://docs.aws.amazon.com/cli/latest/userguide/
- https://forums.aws.amazon.com/forum.jspa?forumID=150
- http://docs.aws.amazon.com/cli/latest/reference/
- http://jmespath.org/
- Boto3 Talk

AWS re:Invent

Please give us your feedback on this presentation

DEV301

amazon web services

Join the conversation on Twitter with #reinvent