Big Data on AWS

Lab Six: Visualizing Using Development Endpoints for Developing Scripts for Glue

Table of Contents

[Lab Six: Visualizing Using Development Endpoints for Developing Scripts for Glue 3](#_Toc509762979)

[Task 1: Create an IAM Role for AWS Glue 3](#_Toc509762980)

[Task 2: Create an IAM Role for Notebooks 4](#_Toc509762981)

[Task 3: Set Up Security Group for Amazon EC2 for a Notebook Server 4](#_Toc509762982)

[Task 4: Set Up Prerequisites for the Development Endpoint 5](#_Toc509762983)

[Task 5: Creating a Development Endpoint for Amazon S3 Data 5](#_Toc509762984)

[Task 6: Set Up an Apache Zeppelin Notebook on Amazon EC2 6](#_Toc509762985)

[Task 7: Connecting to Your Notebook Server on Amazon EC2 7](#_Toc509762986)

[Task 8: Running a Simple Script Fragment in a Notebook Paragraph 7](#_Toc509762987)

[Task 9: Crawl the Data in the Amazon S3 Bucket 8](#_Toc509762988)

[Task 10: Add Boilerplate Script to the Development Endpoint Notebook 8](#_Toc509762989)

[Task 11: Examine the Schemas in the Data Catalog 9](#_Toc509762990)

[Task 12: Filter the Data 10](#_Toc509762991)

[Task 13: Put it all together 10](#_Toc509762992)

[Task 14: Relationalize the Data 11](#_Toc509762993)

# Lab Six: Visualizing Using Development Endpoints for Developing Scripts for Glue

In this lab, to use a development endpoint, we will follow this workflow.

1. Create an AWS Glue development endpoint through the console or API. This endpoint is launched in your virtual private cloud (VPC) with your defined security groups.
2. The console or API can poll the development endpoint until it is provisioned and ready for work. When it's ready, you can connect to the development endpoint to create and test AWS Glue scripts.
3. When you finish debugging and testing on your development endpoint, you can delete it.

Through this lab, you will perform the following tasks:

* Task 1: Create an IAM Role for AWS Glue
* Task 2: Create an IAM Role for Notebooks
* Task 3: Set up a security group for Amazon EC2 for a Notebook Server
* Task 4: Set-up the prerequisites for the development endpoint
* Task 5: Creating a Development Endpoint for Amazon S3 Data
* Task 6: Set Up an Apache Zeppelin Notebook on Amazon EC2
* Task 7: Connecting to Your Notebook Server on Amazon EC2
* Task 8: Running a Simple Script Fragment in a Notebook Paragraph
* Task 9: Crawl the Data in the Amazon S3 Bucket
* Task 10: Add Boilerplate Script to the Development Endpoint Notebook
* Task 11: Examine the Schemas in the Data Catalog
* Task 12: Filter the Data
* Task 13: Put it all together
* Task 14: Relationalize the Data

Note: This lab may incur nominal charges so, be sure to clean up after the lab is over.

## Task 1: Create an IAM Role for AWS Glue

1. Go to IAM Console.
2. In the left navigation pane, choose **Roles**.
3. Choose Create role.
4. For role type, choose **AWS Service**, find and choose **Glue**, and choose **Next: Permissions**.
5. On the **Attach permissions policy** page, choose the policies that contain the required permissions; for example, the AWS managed policy **AWSGlueServiceRole** for general AWS Glue permissions and the AWS managed policy **AmazonS3FullAccess** for access to Amazon S3 resources. Then choose **Next: Review**.
6. For **Role name**, type a name for your role; for example, **AWSGlueServiceRoleDefault**. Create the role with the name prefixed with the string **AWSGlueServiceRole** to allow the role to be passed from console users to the service. AWS Glue provided policies expect IAM service roles to begin with **AWSGlueServiceRole**. Choose **Create Role**.

## Task 2: Create an IAM Role for Notebooks

1. Go to IAM Console.
2. In the left navigation pane, choose **Roles**.
3. Choose Create role.
4. For role type, choose **AWS Service**, find and choose **EC2**, and choose the **EC2** use case, then choose **Next: Permissions**.
5. On the **Attach permissions policy** page, choose the policies that contain the required permissions; for example, **AWSGlueServiceNotebookRole** for general AWS Glue permissions and the AWS managed policy **AmazonS3FullAccess** for access to Amazon S3 resources. Then choose **Next: Review**.
6. For **Role name**, type a name for your role. Create the role with the name prefixed with the string **AWSGlueServiceNotebookRole** to allow the role to be passed from console users to the notebook server. AWS Glue provided policies expect IAM service roles to begin with **AWSGlueServiceNotebookRole**. For example, type **AWSGlueServiceNotebookRoleDefault**. Then choose **Create role**.

## Task 3: Set Up Security Group for Amazon EC2 for a Notebook Server

1. Go to EC2 console.
2. In the left navigation pane, choose **Security Groups**.
3. Either choose an existing security group from the list, or **Create Security Group** to use with your notebook server. The security group that is associated with your development endpoint is also used to create your notebook server.
4. In the security group pane, navigate to the **Inbound** tab.
5. Add inbound rules similar to this:

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Protocol | Port Range | Source |
| SSH | TCP | 22 | 0.0.0.0/0 |
| HTTPS | TCP | 443 | 0.0.0.0/0 |

## Task 4: Set Up Prerequisites for the Development Endpoint

1. In the AWS Glue console, choose **Databases** in the navigation pane, and then choose **Add database**. Name the databaselegislators.
2. Choose **Crawlers**, and then choose **Add crawler**. Name the crawler legislator\_crawler, assign it your AWS Glue role, and then choose **Next**.
3. Leave Amazon S3 as the data store. Under **Crawl data in**, choose **Specified path in another account**. Then in the **Include path** box, type s3://awsglue-datasets/examples/us-legislators/all. Choose **Next**, and then choose **Next** again to confirm that you don't want to add another data store. Then choose **Next** to confirm that this crawler will be run on demand.
4. For **Database**, choose the legislators database. Choose **Next**, and then choose **Finish** to complete the creation of the new crawler.
5. Choose **Crawlers** in the navigation pane again. Select the check box next to the new legislator\_crawler crawler, and choose **Run crawler**.
6. Choose **Databases** in the navigation pane. Choose the legislators database, and then choose **Tables in legislators**. You should see six tables created by the crawler in your Data Catalog, containing metadata that the crawler retrieved.

## Task 5: Creating a Development Endpoint for Amazon S3 Data

1. In the AWS Glue console, choose **Dev endpoints**. Choose **Add endpoint**.
2. Specify an endpoint name, such as **demo-endpoint**.
3. Choose an IAM role with name AWSGlueServiceRoleDefault. Choose Next.
4. In Networking, leave Skip networking information selected, and choose Next.
5. In **SSH Public Key**, enter a public key generated by an SSH key generator program (do not use an Amazon EC2 key pair). Save the corresponding private key to later connect to the development endpoint using SSH. Choose **Next**.

**Note:** When generating the key on Microsoft Windows, use a current version of PuTTYgen and paste the public key into the AWS Glue console from the PuTTYgen window. Generate an **RSA** key. Do not upload a file with the public key, instead use the key generated in the field **Public key for pasting into OpenSSH authorized\_keys file**. The corresponding private key (.ppk) can be used in PuTTY to connect to the development endpoint. To connect to the development endpoint with SSH on Windows, convert the private key from .ppk format to OpenSSH .pem format using the PuTTYgen **Conversion** menu.

1. In **Review**, choose **Finish**. After the development endpoint is created, wait for its provisioning status to move to **READY**.
2. If you don't already have a bucket, create a Bucket in Amazon S3 where you can save output from sample ETL scripts.

## Task 6: Set Up an Apache Zeppelin Notebook on Amazon EC2

1. On the AWS Glue console, choose **Dev endpoints** to go to the development endpoints list.
2. Choose an endpoint by selecting the box next to it. Then choose **Actions**, and choose **Create notebook server**.
3. To host the notebook server, an Amazon EC2 instance is spun up using an AWS CloudFormation stack on your development endpoint, and a Zeppelin notebook HTTP server is started on port 443.
4. Enter an AWS CloudFormation stack server name such as demo-cf, using only alphanumeric characters and hyphens.
5. Choose an IAM role with the name **AWSGlueServiceNotebookRole**.
6. Choose an Amazon EC2 key pair that you have generated on the Amazon EC2 console ([https://console.aws.amazon.com/ec2/](https://console.aws.amazon.com/ec2/" \t "_blank)), or choose **Create EC2 key pair** to generate a new one. Remember where you have downloaded and saved the private key portion of the pair. This key pair is different from the SSH key you used when creating your development endpoint (the keys that Amazon EC2 uses are 2048-bit SSH-2 RSA keys).
7. It is to generally a good practice to ensure that the private-key file is write-protected so that it is not accidentally modified. On macOS and Linux systems, do this by opening a terminal and entering chmod 400 private-key-file path. On Windows, open the console and enter attrib -r private-key-file path.
8. Choose a user name and password to access your Zeppelin notebook.
9. Choose an Amazon S3 path for your notebook state to be stored in.
10. Choose **Create**.
11. You can view the status of the AWS CloudFormation stack in the AWS CloudFormation console **Events** tab ([https://console.aws.amazon.com/cloudformation](https://console.aws.amazon.com/cloudformation/" \t "_blank)).
12. You can view the Amazon EC2 instances created by AWS CloudFormation in the Amazon EC2 console ([https://console.aws.amazon.com/ec2/](https://console.aws.amazon.com/ec2/" \t "_blank)).
13. Search for instances that are tagged with the key name **aws-glue-dev-endpoint** and value of the name of your development endpoint.

After the notebook server is created, its status changes to **CREATE\_COMPLETE** in the Amazon EC2 console. Details about your server also appear in the development endpoint details page. When the creation is complete, you can connect to a notebook on the new server.

**Note**: For any notebook server that you create that is associated with a development endpoint, you manage it. Therefore, if you delete the development endpoint, to delete the notebook server, you must delete the AWS CloudFormation stack on the AWS CloudFormation console.

## Task 7: Connecting to Your Notebook Server on Amazon EC2

1. In the AWS Glue console, choose Dev endpoints to navigate to the development endpoints list. Choose the name of the development endpoint for which you created a notebook server. Choosing the name opens its details page.
2. At the bottom of the **Endpoint details** page, copy the URL labeled **Notebook Server URL**.
3. Open a web browser, and paste in the notebook server URL. This lets you access the server using HTTPS on port 443. Your browser may not recognize the server's certificate, in which case you have to override its protection and proceed anyway.
4. Log in to Zeppelin using the user name and password that you provided when you created the notebook server.

## Task 8: Running a Simple Script Fragment in a Notebook Paragraph

1. Choose Create new note and name it Legislators. Confirm spark as the Default Interpreter.
2. You can verify that your notebook is now set up correctly by typing the statement spark.version and running it. This returns the version of Apache Spark that is running on your notebook server.
3. Type the following script into the next paragraph in your notebook and run it. This script reads metadata from the persons\_json table that your crawler created, creates a DynamicFrame from the underlying data, and displays the number of records and the schema of the data.

%pyspark

import sys

from pyspark.context import SparkContext

from awsglue.context import GlueContext

from awsglue.transforms import \*

from awsglue.utils import getResolvedOptions

# Create a Glue context

glueContext = GlueContext(SparkContext.getOrCreate())

# Create a DynamicFrame using the 'persons\_json' table

persons\_DyF = glueContext.create\_dynamic\_frame.from\_catalog(database="legislators", table\_name="persons\_json")

# Print out information about this data

print "Count: ", persons\_DyF.count()

persons\_DyF.printSchema()

1. Run the script

## Task 9: Crawl the Data in the Amazon S3 Bucket

1. Sign in to the AWS Management Console, and open the AWS Glue console at [https://console.aws.amazon.com/glue/](https://console.aws.amazon.com/glue/" \t "_blank)
2. Create a new crawler that can crawl thes3://awsglue-datasets/examples/us-legislators/all dataset into a database named legislators in the AWS Glue Data Catalog.
3. Run the new crawler, and then check the legislators database.

The crawler creates the following metadata tables:

persons\_json

memberships\_json

organizations\_json

events\_json

areas\_json

countries\_r\_json

This is a semi-normalized collection of tables containing legislators and their histories.

## Task 10: Add Boilerplate Script to the Development Endpoint Notebook

1. Paste the following boilerplate script into the development endpoint notebook to import the AWS Glue libraries that you need, and set up a single GlueContext:
2. Paste the following boilerplate script into the development endpoint notebook to import the AWS Glue libraries that you need, and set up a single GlueContext:

import sys

from awsglue.transforms import \*

from awsglue.utils import getResolvedOptions

from pyspark.context import SparkContext

from awsglue.context import GlueContext

from awsglue.job import Job

glueContext = GlueContext(SparkContext.getOrCreate())

## Task 11: Examine the Schemas in the Data Catalog

Next, you can easily examine the schemas that the crawler recorded in the AWS Glue Data Catalog.

1. For example, to see the schema of the persons\_json table, add the following in your notebook:

persons = glueContext.create\_dynamic\_frame.from\_catalog(

database="legislators",

table\_name="persons\_json")

print "Count: ", persons.count()

persons.printSchema()

1. Each person in the table is a member of some US congressional body.
2. To view the schema of the memberships\_json table, type the following:

memberships = glueContext.create\_dynamic\_frame.from\_catalog(

database="legislators",

table\_name="memberships\_json")

print "Count: ", memberships.count()

memberships.printSchema()

1. The organizations are parties and the two chambers of Congress, the Senate and House of Representatives.
2. To view the schema of the organizations\_json table, type the following:

orgs = glueContext.create\_dynamic\_frame.from\_catalog(

database="legislators",

table\_name="organizations\_json")

print "Count: ", orgs.count()

orgs.printSchema()

## Task 12: Filter the Data

Next, keep only the fields that you want, and rename id to org\_id. The dataset is small enough that you can view the whole thing.

1. The toDF() converts a DynamicFrame to an Apache Spark DataFrame, so you can apply the transforms that already exist in Apache Spark SQL:

orgs = orgs.drop\_fields(['other\_names',

'identifiers']).rename\_field(

'id', 'org\_id').rename\_field(

'name', 'org\_name')

orgs.toDF().show()

1. Type the following to view the organizations that appear in memberships:

memberships.select\_fields(['organization\_id']).toDF().distinct().show()

## Task 13: Put it all together

Now, use AWS Glue to join these relational tables and create one full history table of legislator memberships and their corresponding organizations.

1. First, join persons and memberships on id and person\_id.
2. Next, join the result with orgs on org\_id and organization\_id.
3. Then, drop the redundant fields, person\_id and org\_id.

You can do all these operations in one (extended) line of code:

l\_history = Join.apply(orgs,

Join.apply(persons, memberships, 'id', 'person\_id'),

'org\_id', 'organization\_id').drop\_fields(['person\_id', 'org\_id'])

print "Count: ", l\_history.count()

l\_history.printSchema()

1. You now have the final table that you can use for analysis. You can write it out in a compact, efficient format for analytics—namely Parquet—that you can run SQL over in AWS Glue, Amazon Athena, or Amazon Redshift Spectrum.
2. The following call writes the table across multiple files to support fast parallel reads when doing analysis later:

glueContext.write\_dynamic\_frame.from\_options(frame = l\_history,

connection\_type = "s3",

connection\_options = {"path": "s3://glue-sample-target/output-dir/legislator\_history"},

format = "parquet")

1. To put all the history data into a single file, you must convert it to a data frame, repartition it, and write it out:

s\_history = l\_history.toDF().repartition(1)

s\_history.write.parquet('s3://glue-sample-target/output-dir/legislator\_single')

1. Or, if you want to separate it by the Senate and the House:

l\_history.toDF().write.parquet('s3://glue-sample-target/output-dir/legislator\_part',

partitionBy=['org\_name'])

## Task 14: Relationalize the Data

1. AWS Glue makes it easy to write the data to relational databases like Amazon Redshift, even with semi-structured data.
2. It offers a transform relationalize, which flattens DynamicFrames no matter how complex the objects in the frame might be.
3. Using the l\_history DynamicFrame in this example, pass in the name of a root table (hist\_root) and a temporary working path to relationalize. This returns a DynamicFrameCollection. You can then list the names of the DynamicFrames in that collection:

dfc = l\_history.relationalize("hist\_root", "s3://glue-sample-target/temp-dir/") dfc.keys()

1. The following is the output of the keys call:

[u'hist\_root', u'hist\_root\_contact\_details', u'hist\_root\_links', u'hist\_root\_other\_names', u'hist\_root\_images', u'hist\_root\_identifiers']

Relationalize broke the history table out into six new tables: a root table that contains a record for each object in the DynamicFrame, and auxiliary tables for the arrays.

Array handling in relational databases is often suboptimal, especially as those arrays become large. Separating the arrays into different tables makes the queries go much faster.

1. Next, look at the separation by examining contact\_details:

l\_history.select\_fields('contact\_details').printSchema() dfc.select('hist\_root\_contact\_details').toDF().where("id = 10 or id = 75").orderBy(['id','index']).show()

Congratulations!! You have completed the lab.