Talend connect to AWS and Snowflake Tutorial

In this tutorial we will be examining connections from Talend to various AWS services and the Snowflake Data Warehouse package. In AWS we will use S3, SQS and EC2 services together with Lambda functions. To demonstrate the usage the following scenario will be modelled.

A retail company employs staff at several different sites which open 12 hours daily with a staffing FTE (full time equivalent) level of 5. Each store has a pool of 20 staff who work 6-hour shifts and to fulfil the FTE, any 10 of the 20 staff will be required each day. The timesheets for each day are produced by an external system that periodically sends a data file in JSON format to an Amazon S3 bucket owned by the company, who will then load the information from this file into their Snowflake database, for further analysis.

In the example a single timesheet file for each week will be produced for all stores in a simulated process, as is the creation of staff records. These simulations are useful for demonstrating Talend concepts.

Requirements

To undertake this assignment, the following software and accounts are required.

- Talend Big Data Studio community edition version 7.3 or later.
- Amazon Web Services account, all examples are free tier eligible
- Snowflake account, trial version used in this tutorial.
- Java 8 JRE for installation onto AWS EC2 instance.

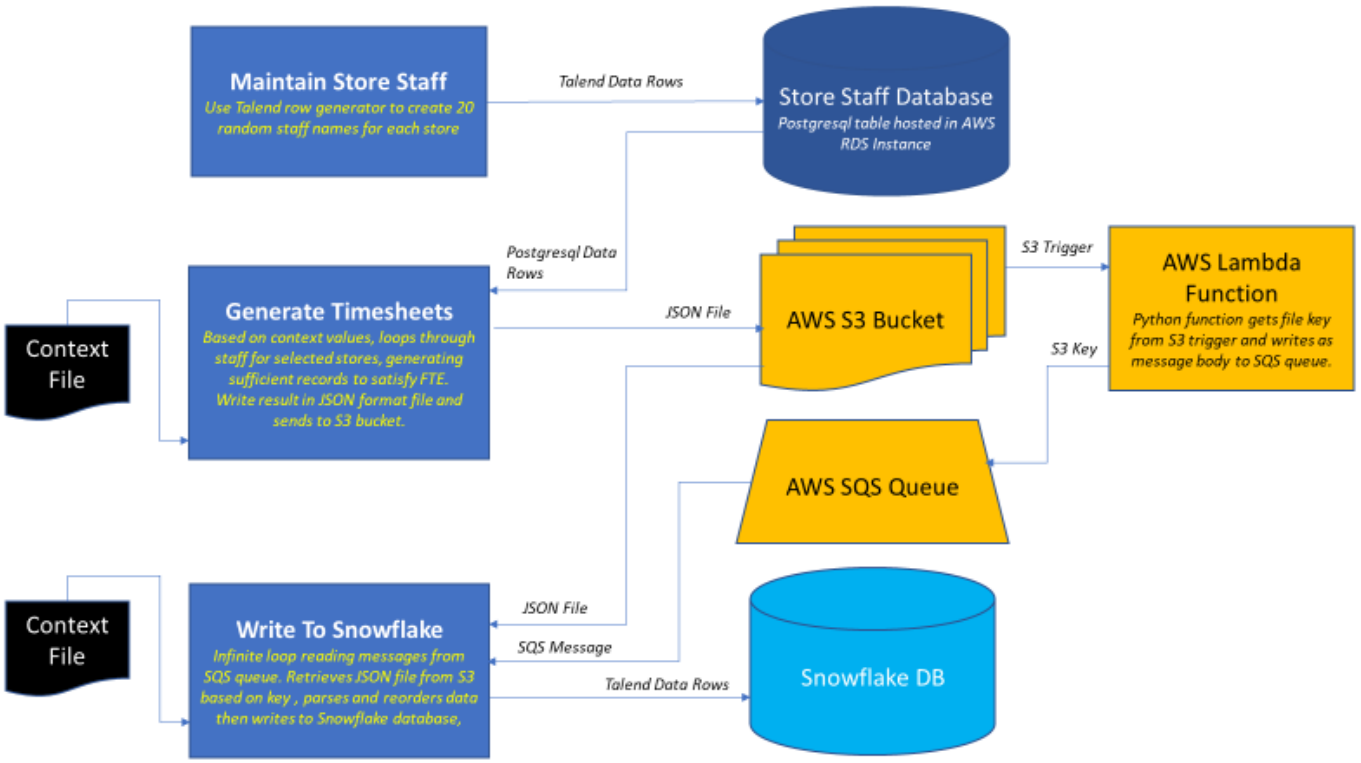
Design Overview

Store staff records will be created by a Talend job and stored in a Postgresql table hosted via RDS an AWS. A second Talend job running on an EC2 instance will extract data from the staff database and use it to simulate the generation of timesheets. These will be written to a JSON format file and be copied to an S3 bucket. An AWS Lambda function written in Python responds to an S3 trigger event raised by placement of the file into the bucket and writes the file key to the body of a message, which is placed on an AWS SQS queue. A final Talend job also running on EC2 responds to the SQS message and retrieves the file key from the message body.

The key is used to download the JSON file from S3 and its contents are parsed and reordered before being written to a table in Snowflake. Both Talend jobs running on the EC2 instance use context files to load the initial variables

The use of the Lambda function allows serverless technology to be used to automate the snowflake load on receipt of a timesheet file.

The following illustration shows the design concept.



Task List

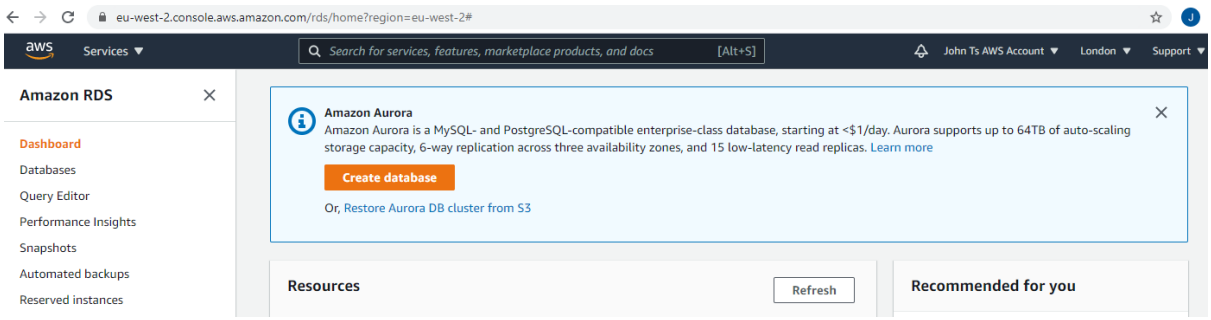
Before generating the Talend jobs for this project there are several configuration tasks to be undertaken. The following table provides a check list of these:

No	Task	Description
1	Create Postgresql instance on AWS RDS	Working in the AWS console for RDS, create a new Postgresql instance.
2	Add database and table to Postgres	Using the instance created previously create a database and table within it to hold details of store staff.
3	Setup Snowflake schema.	Create a new database in Snowflake and add a table to store staff hours.
4	Create AWS S3 bucket	Create a bucket in the S3 console for timesheet files to be loaded into.
5	Generate AWS Keys	Generate AWS key pair and download for use in connecting to S3 and SQS.
6	Create an EC2 Instance	Create AWS EC2 instance for Talend standalone jobs to run on.
7	Install Java 8 JRE onto EC2	Install the java runtime onto the EC2 instance created previously. This will allow Talend standalone jobs to run on the server.
8	Create S3 role for EC2	Create a role with access to S3 buckets and assign the role to the EC2 instance to allow applications access to S3.
9	Create AWS SQS queue	Create a FIFO queue in AWS which will be written to by Lambda and consumed by Talend.
10	Create AWS Lambda function	Create a serverless Lambda function in AWS triggered by S3 input that write file key from the trigger into an SQS message. Code will be written in Python, directly into AWS console.
11	Create job to generate store staff	Create Talend job to generate store staff.
12	Create job to generate timesheets	Talend job to create timesheets and write as a JSON file to the S3 bucket. This will trigger the Lambda, writing a message to SQS.
13	Create job to write to Snowflake	Job that loops indefinitely, reading SQS queue and retrieving data from S3. Output goes to Snowflake database.
14	Deploy jobs to EC2	Deploy the 2 Talend jobs that will be ran on the EC2 server.

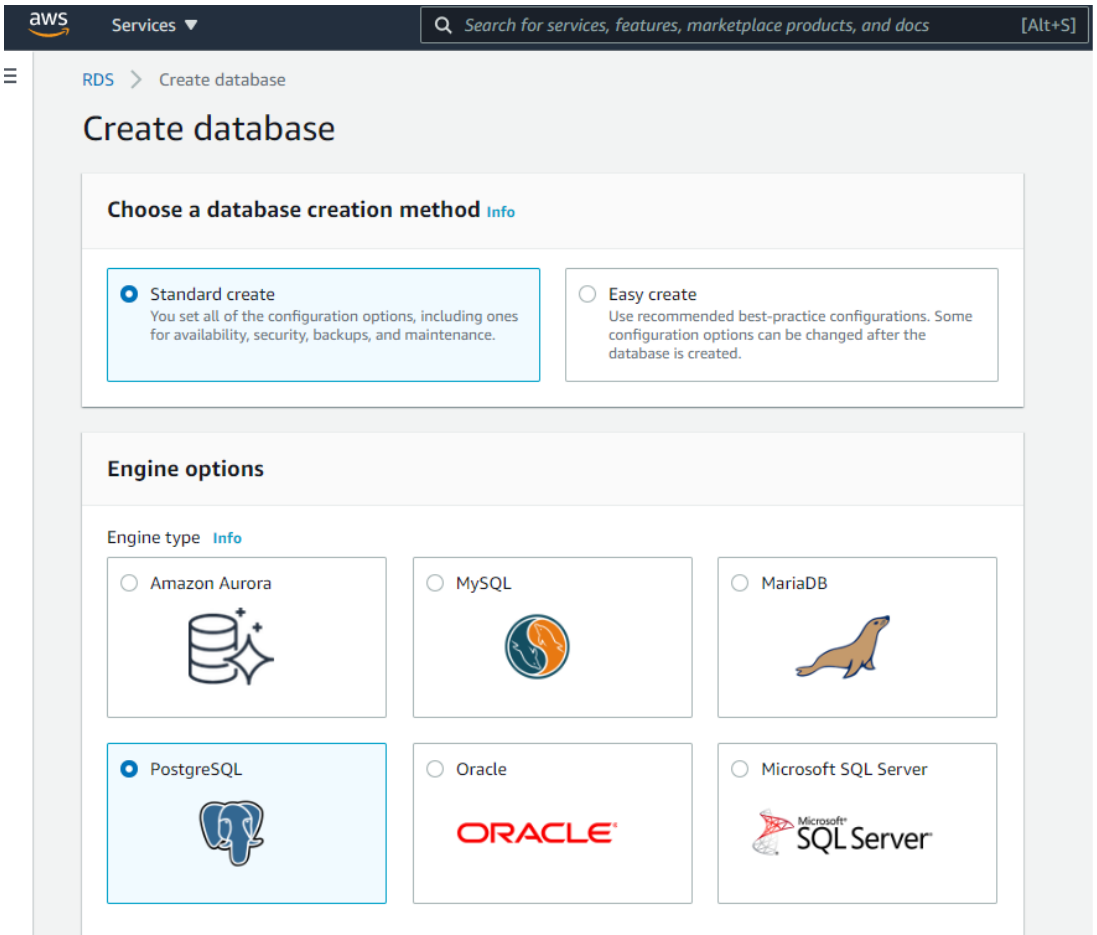
Each task will be examined in turn.

1. Create Postgresql instance on AWS RDS

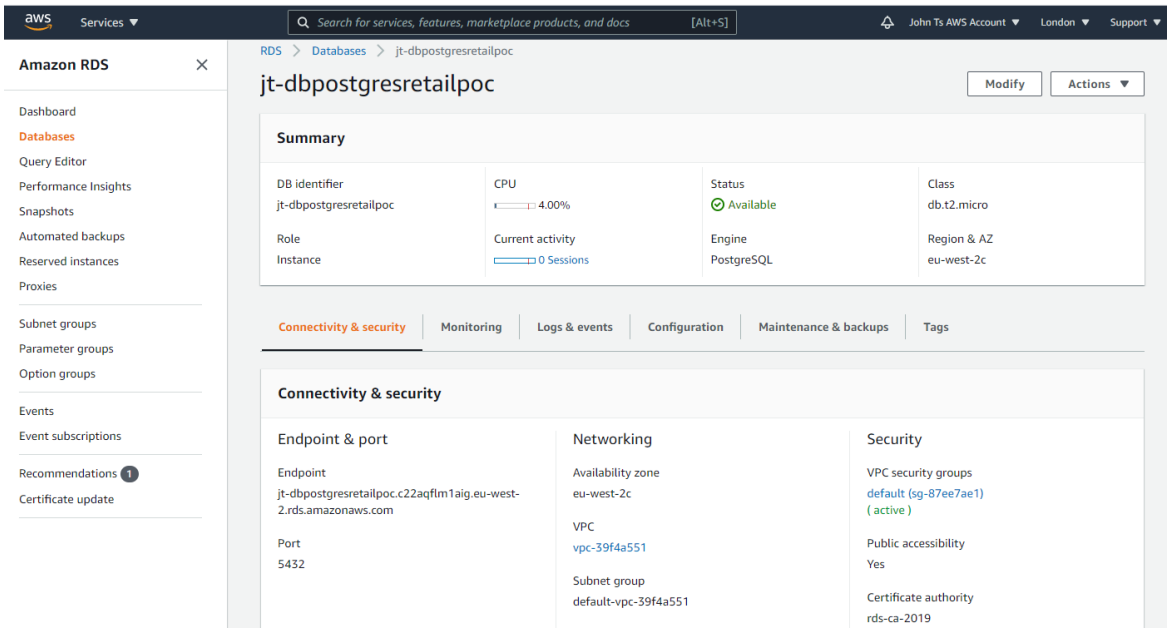
To add a new PostgreSQL database into AWS RDS an instance must be created. Log onto AWS management console and enter the RDS option. Click on the **Create Database** button.



In the Create Database dialog select **PostgreSQL** from the Engine options box and check the box to use the **Free tier template**.

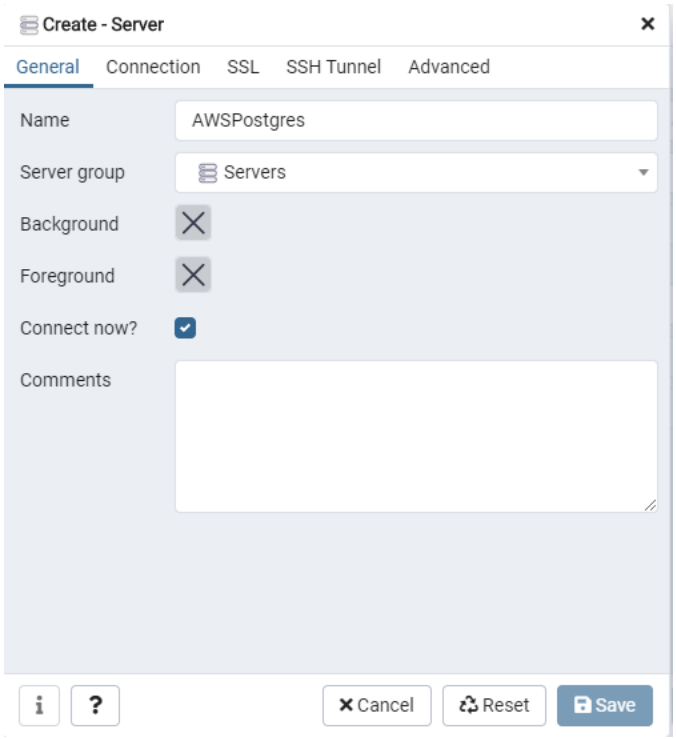


Give the instance a name, in this case jt-dbpostgresretailpoc, accept all defaults and click **Create**. The instance will be created, and this can be checked by selecting **Databases** from the left-hand side menu and clicking on the **instance name**. Information regarding the instance including endpoint and port details will be displayed.



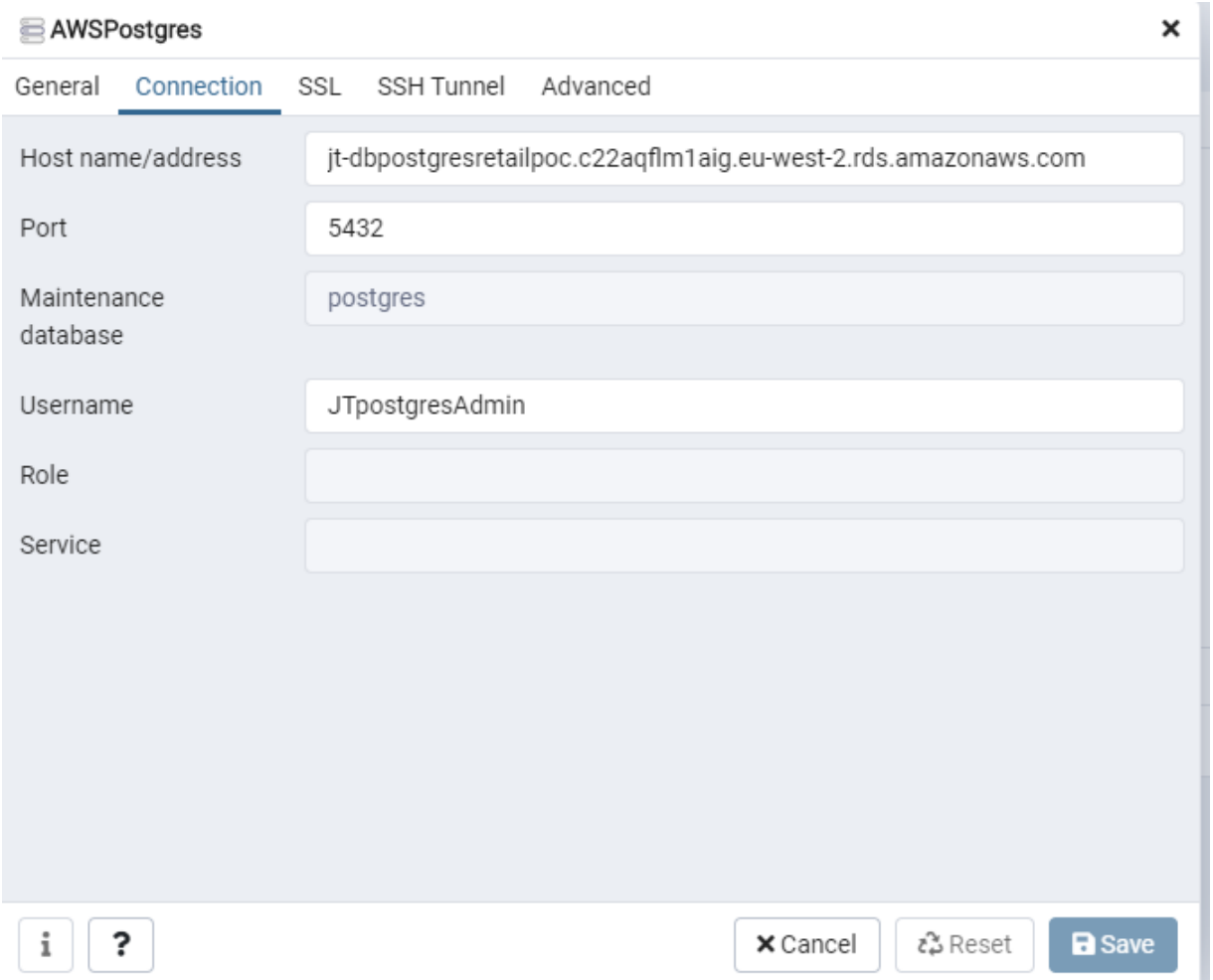
2. Add database and table to Postgres

The previous step has launched a PostgreSQL instance in AWS, but we now need to create a database and table within the instance. This is achieved by connecting to the instance from outside AWS by using the PGAdmin tool, the standard IDE for PostgreSQL.

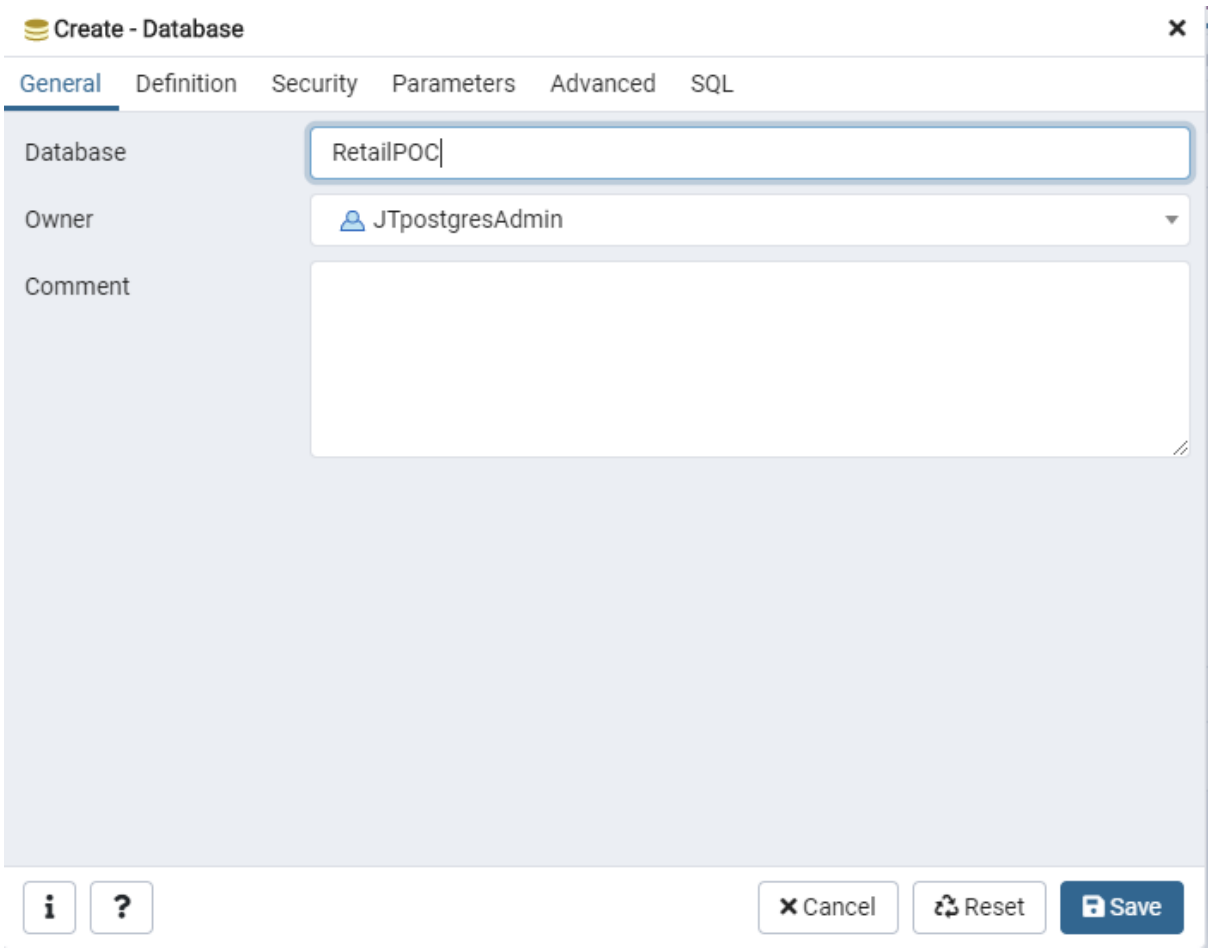


In the PGAdmin browser pain right click on **Servers** and select **Create > Server**. In the first tab of the dialog box enter the server name AWSPostgres.

Select **Connection** from the menu and enter the end point as defined in the AWS RDS instance screen from the previous section, confirming that the port number is also correct. The maintenance database can be left at the default value of postgres and the username/password should be the ones defined in AWS.



Working again in the Browser pane, expand the AWSPostgres server and right click on the **Databases** node. Select **Create > Database** from the menu and name it RetailPOC prior to saving.



Right click the **RetailPOC** database in the Browser and select **Query Tool** from the menu.

Copy the following SQL statement into the query window then execute it:

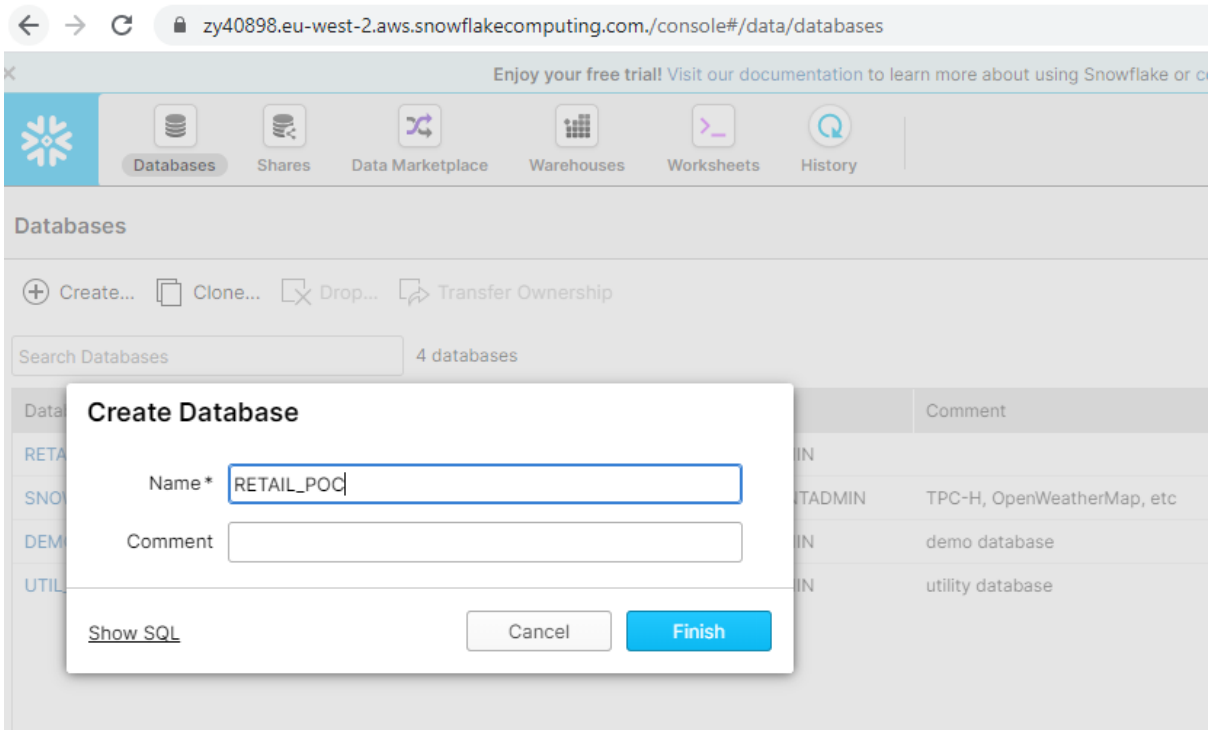
```
CREATE TABLE public.tbstorestaff
(
    employee_no integer NOT NULL,
    store_code integer,
    first_name text COLLATE pg_catalog."default",
    last_name text COLLATE pg_catalog."default",
    CONSTRAINT tbstorestaff_pkey PRIMARY KEY (employee_no)
)
```

The table structure should now be visible in the Browser hierarchy as below:

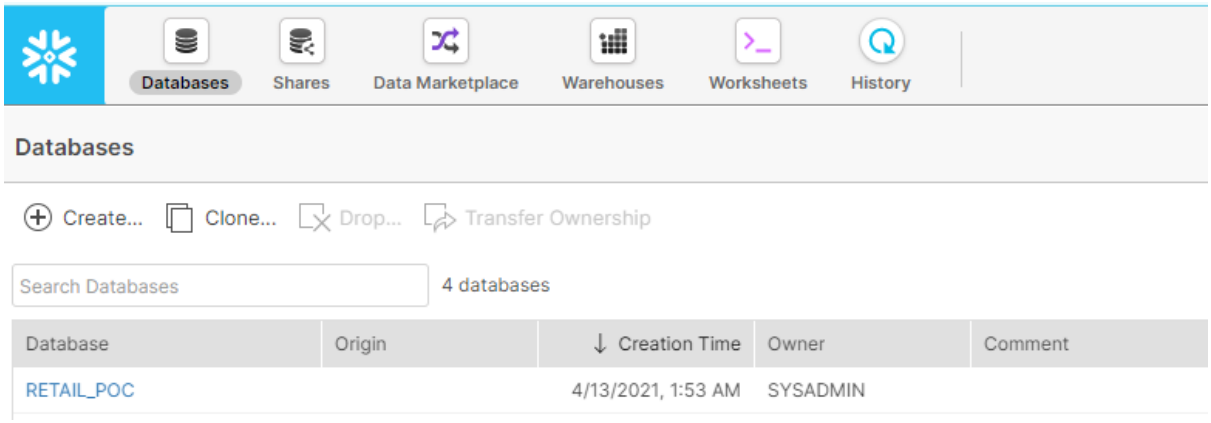
The PostgreSQL table is now ready to receive data.

3. Setup Snowflake schema.

Log in to the Snowflake account and select the Databases option from the ribbon menu. Click on **Create** and enter the name of the database RETAIL_POC in the name field and press **Finish**.



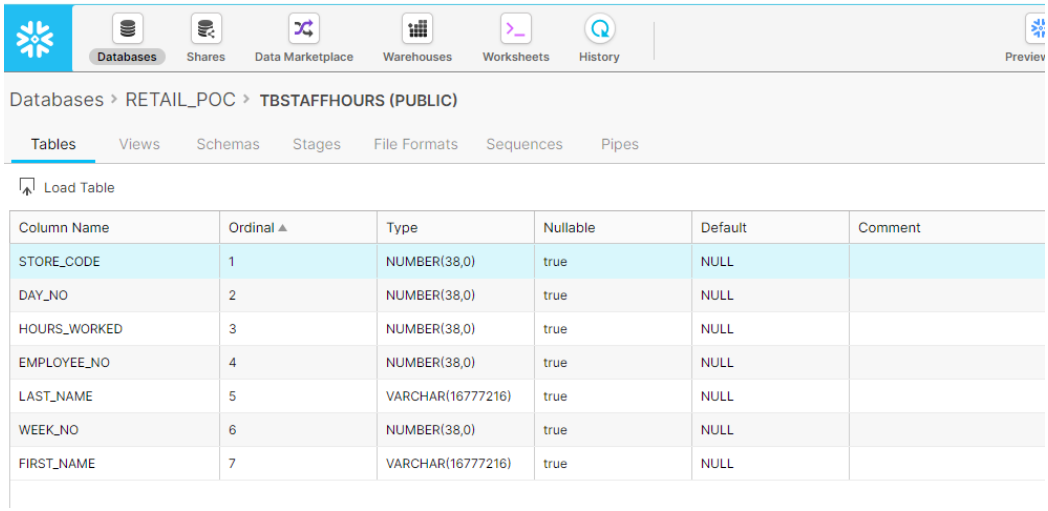
The database will now be shown in the list as below.



Click the **Snowflake icon** at the left-hand end of the ribbon to open a query window. Select the RETAIL_POC database and the PUBLIC schema, then paste the following code into the query window:

```
CREATE OR REPLACE TABLE "RETAIL_POC"."PUBLIC"."TBSTAFFHOURS" (
        STORE_CODE NUMBER(38,0),
        DAY_NO NUMBER(38,0),
        HOURS_WORKED NUMBER(38,0),
        EMPLOYEE_NO NUMBER(38,0),
        LAST_NAME VARCHAR(16777216),
        WEEK_NO NUMBER(38,0),
        FIRST_NAME VARCHAR(16777216)
);
```

Run the query which will create the table. Selecting the Databases option from the menu will now display the TBSTAFFHOURS table. Clicking on the table name should display the definition.



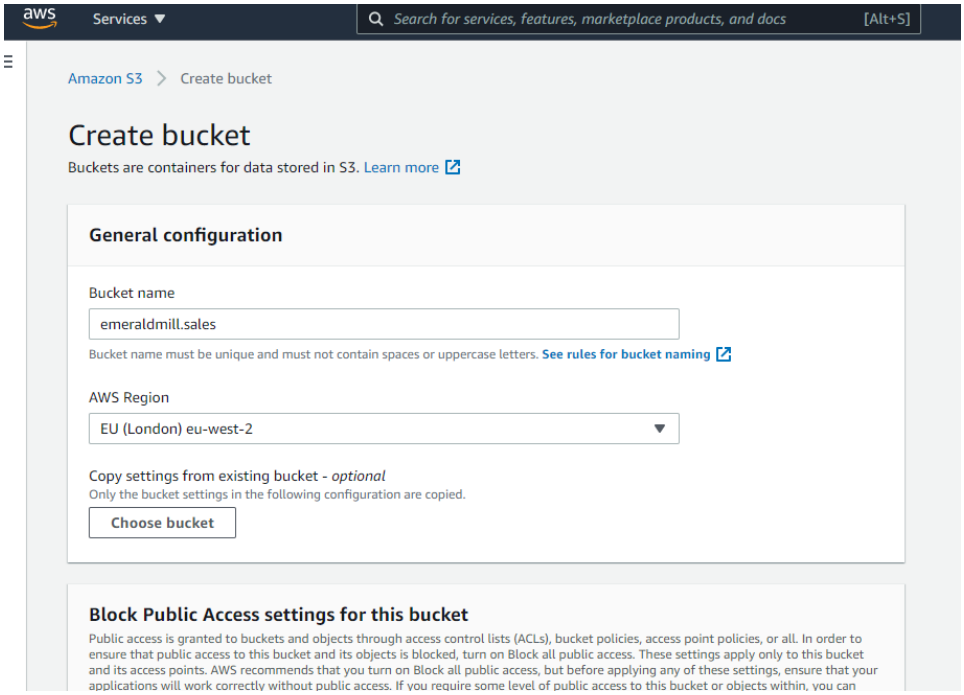
Column Name	Ordinal	Type	Nullable	Default	Comment
STORE_CODE	1	NUMBER(38,0)	true	NULL	
DAY_NO	2	NUMBER(38,0)	true	NULL	
HOURS_WORKED	3	NUMBER(38,0)	true	NULL	
EMPLOYEE_NO	4	NUMBER(38,0)	true	NULL	
LAST_NAME	5	VARCHAR(16777216)	true	NULL	
WEEK_NO	6	NUMBER(38,0)	true	NULL	
FIRST_NAME	7	VARCHAR(16777216)	true	NULL	

Snowflake is now ready to receive the test data. Since we will be accessing the table directly using our Snowflake account there is no need to worry about permissions. In the real world it would be necessary to set up the correct access.

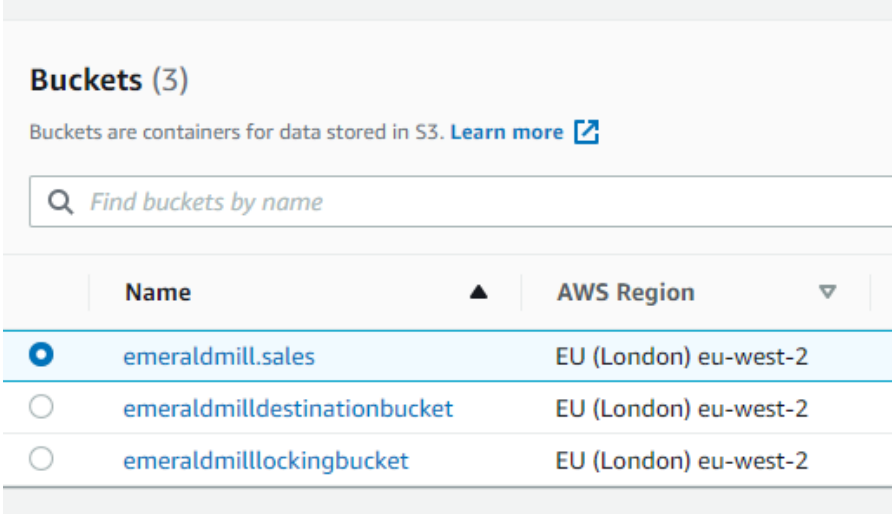
4. Create AWS S3 bucket

An S3 bucket is created to store the JSON files simulating employee hours worked sent from an external application. Since access will be controlled by a key pair, public access is not required and the setup will be relatively simple, predominately using default values for security.

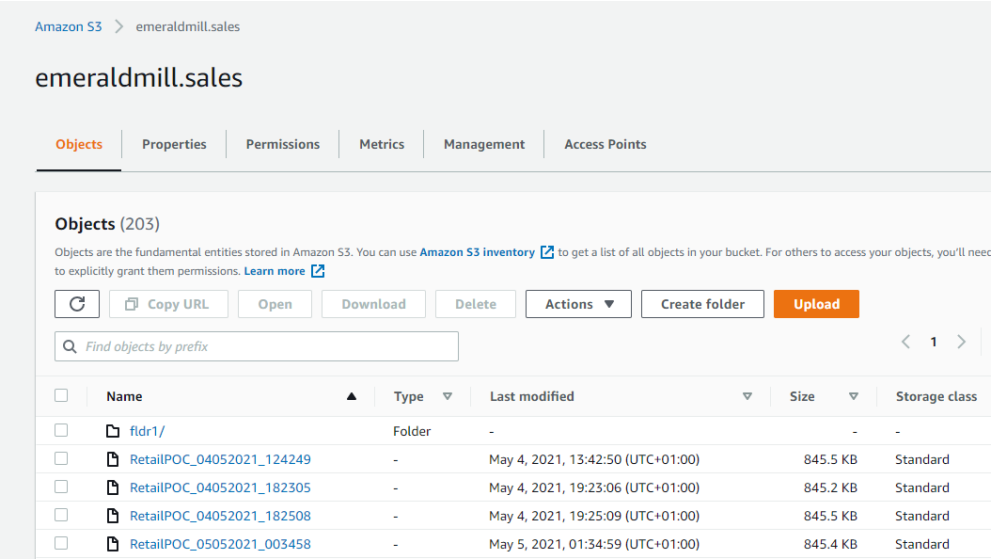
Within AWS navigate to the S3 console and click **Create Bucket**. Give the bucket a unique name and everything else can be left as default. My bucket name is emeraldmill.sales but any unique name is fine. Press the **Create Bucket** button once complete.



The bucket will be created and will appear in your list of buckets.



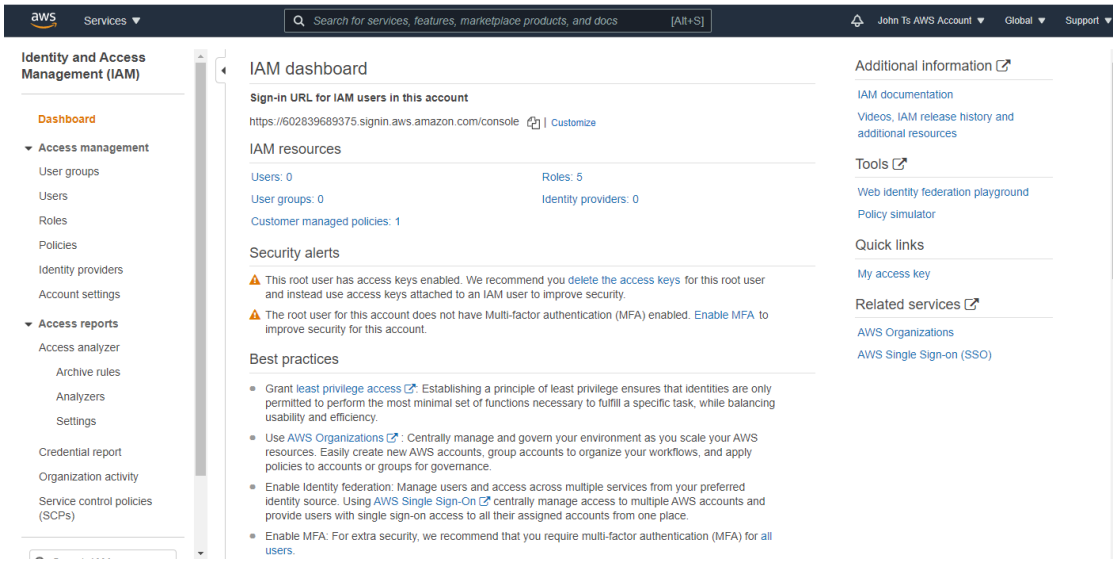
Clicking on the name will take you to the property pages and display the list of objects currently stored in the bucket. Initially this will be empty, but the illustration below shows the bucket after some files have been loaded.



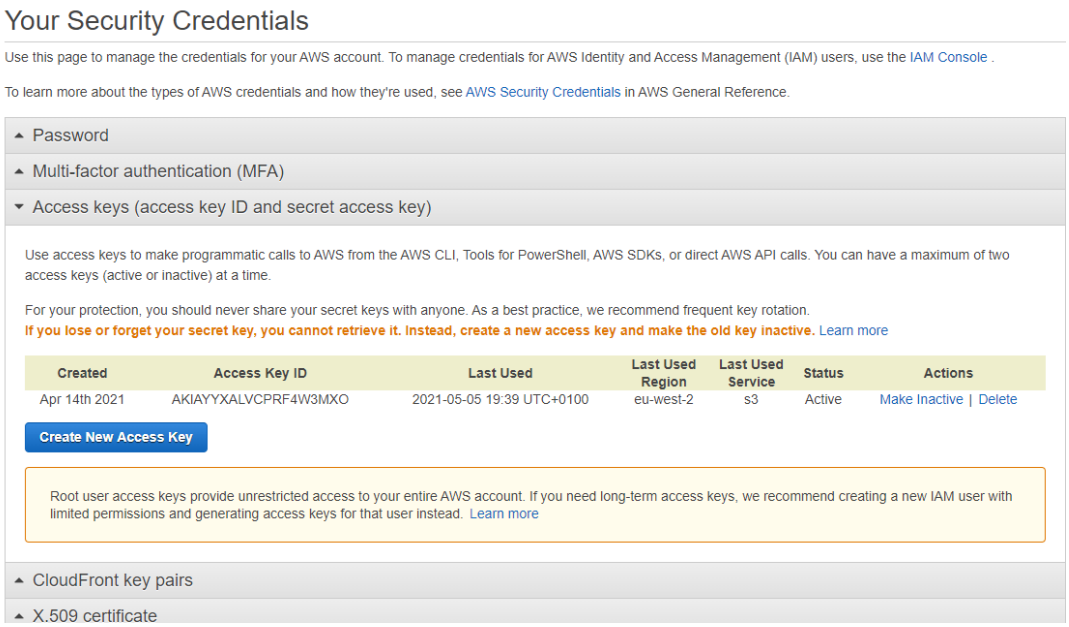
5. Generate AWS Keys

Authentication for the S3 bucket in this POC will use public key cryptography. A key pair generated withing AWS will grant root user access to AWS services. In practice this would be a serious security weakness and a user with the minimum clearance to perform necessary tasks should be created. A key pair against this more restricted user could then be generated limiting access to just necessary areas.

To generate an access key, go to the AWS IAM dashboard and click on the **My access key** link on the right-hand side of the screen.

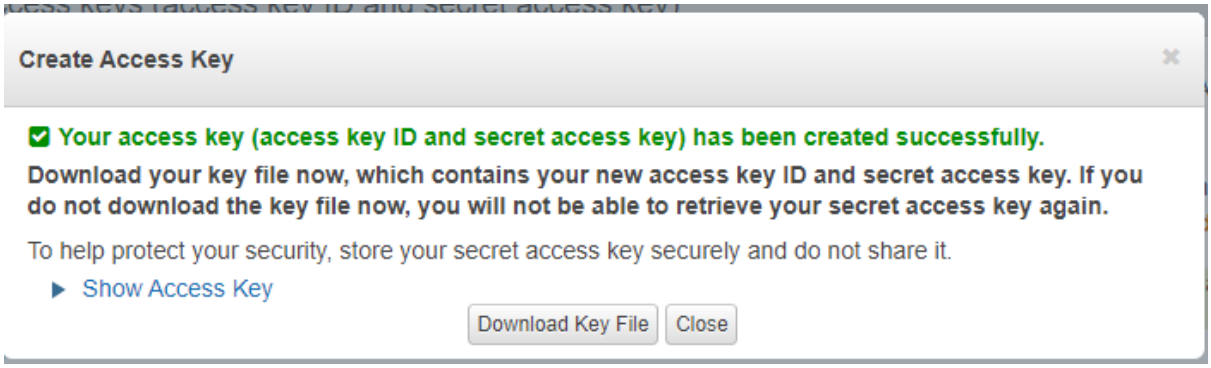


This takes you to the security credentials page. Expand the **Access Keys** section and click on **Create New Access Key**.



Another way of getting to this screen is from the drop-down account menu on the ribbon at the top of all console screens. Select the My Security Credentials option.

The key pair will be created and can be downloaded in a text file. Keep this file in a secure place so the key values can be used later.

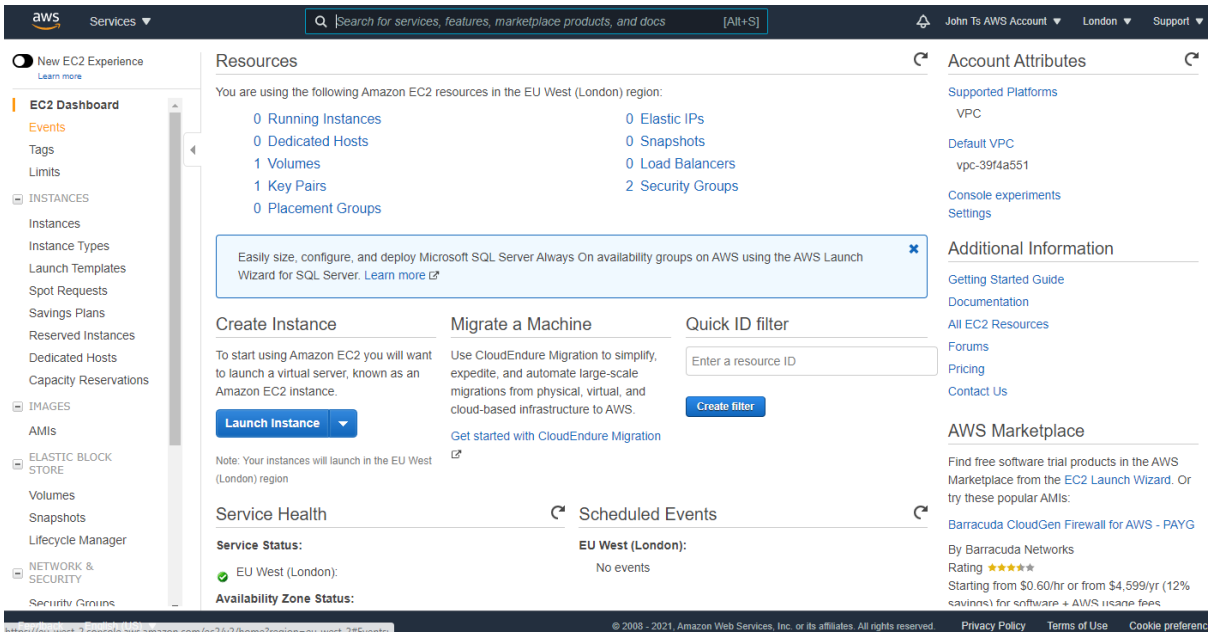


It is common practice to have several keys created with all but one set de-activated. The active key is then changed on a cyclic basis to form an additional layer of security. This is known as key rotation.

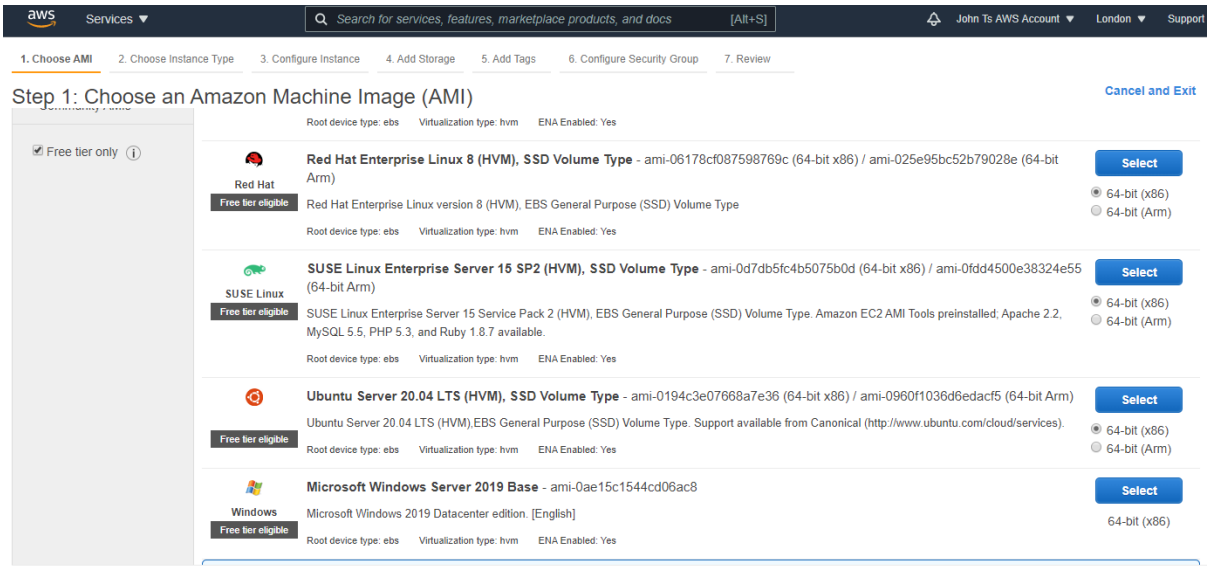
6. Create an EC2 Instance

Two of the Talend jobs will run as standalone jobs on a dedicated server. Creation of timesheet files will run on demand whereas the consumption of these files is a process that loops continuously once initiated. The server will be an AWS EC2 instance and the only pre-requisite will be installation of Java 8. There is no requirement for Talend Studio to be installed as all dependencies are included in the standalone jobs which build in a similar way to a fat jar.

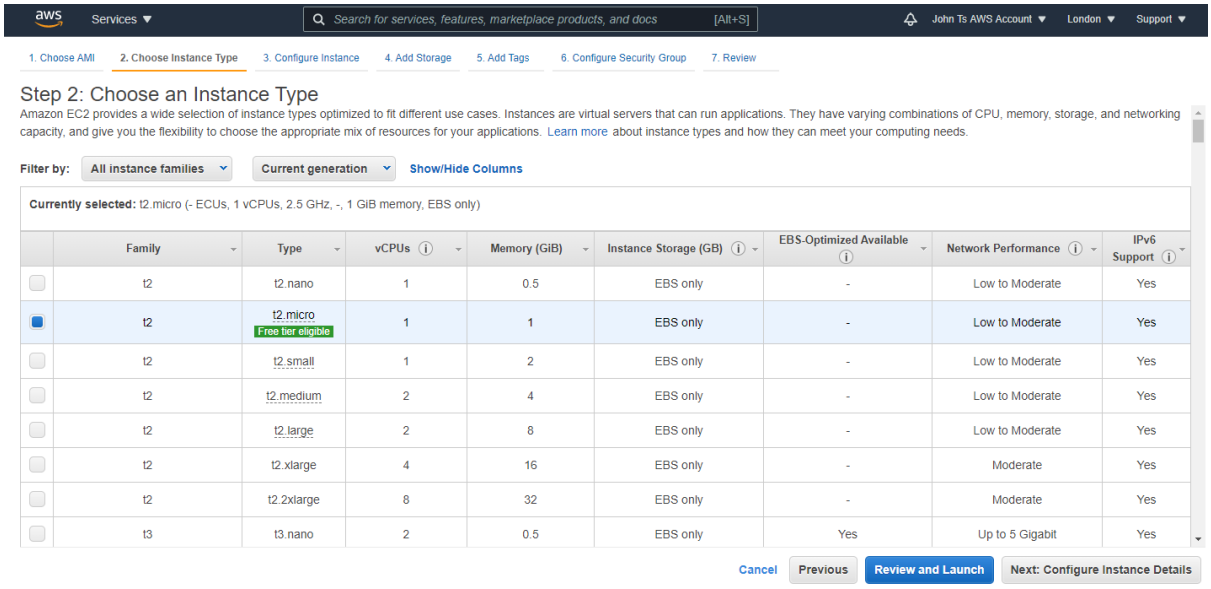
From the AWS EC2 console click on the **Launch Instance** button to create a new virtual server.



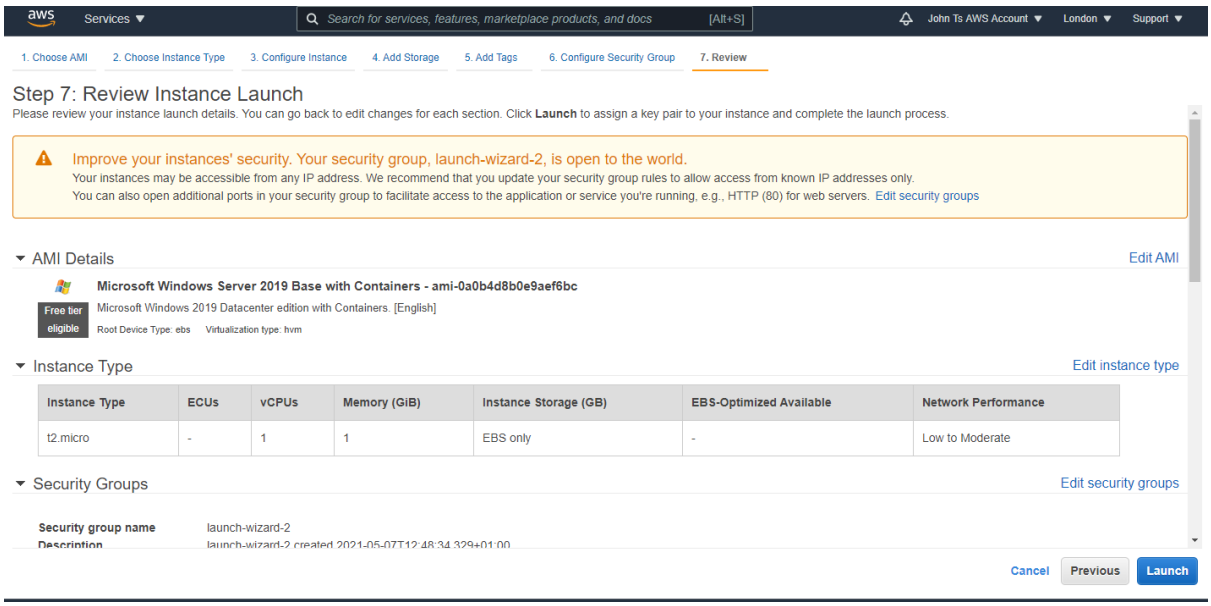
The first stage is to choose the Amazon Machine Image (AMI) on which the new server will be based. Click the Free tier only option on the left-hand pane and scroll down to the **Microsoft Windows Server 2019 Base** option and select.



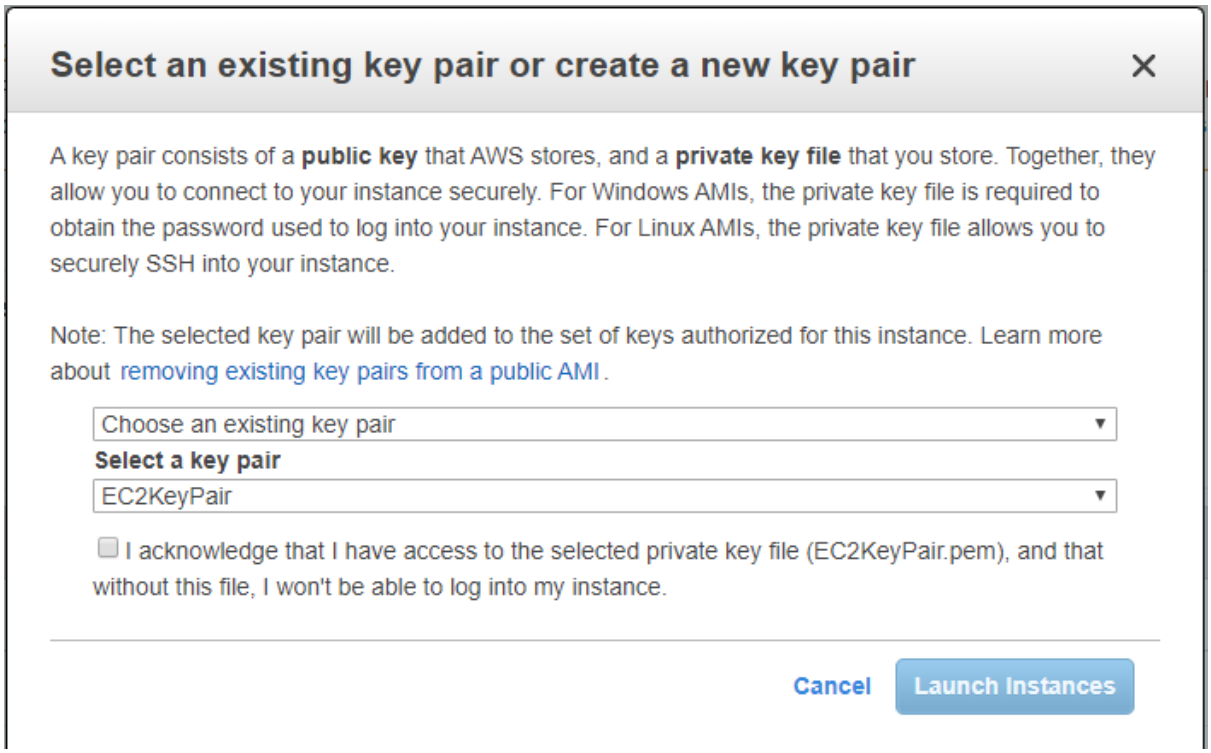
The instance type for Free Tier eligibility will already be selected so click **Review and Launch** to initiate the instance.



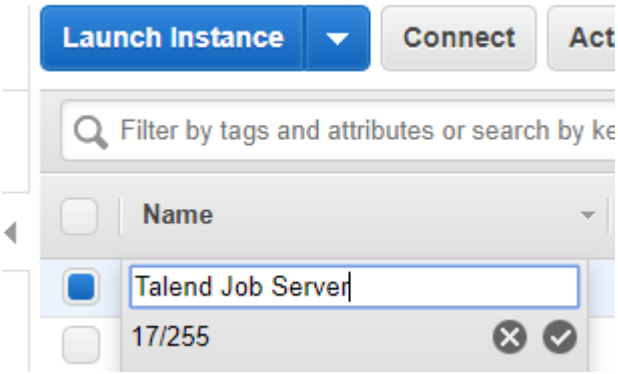
The details of the instance will be displayed. For this proof of concept, the defaults are all fine so click **Launch** to start the server.



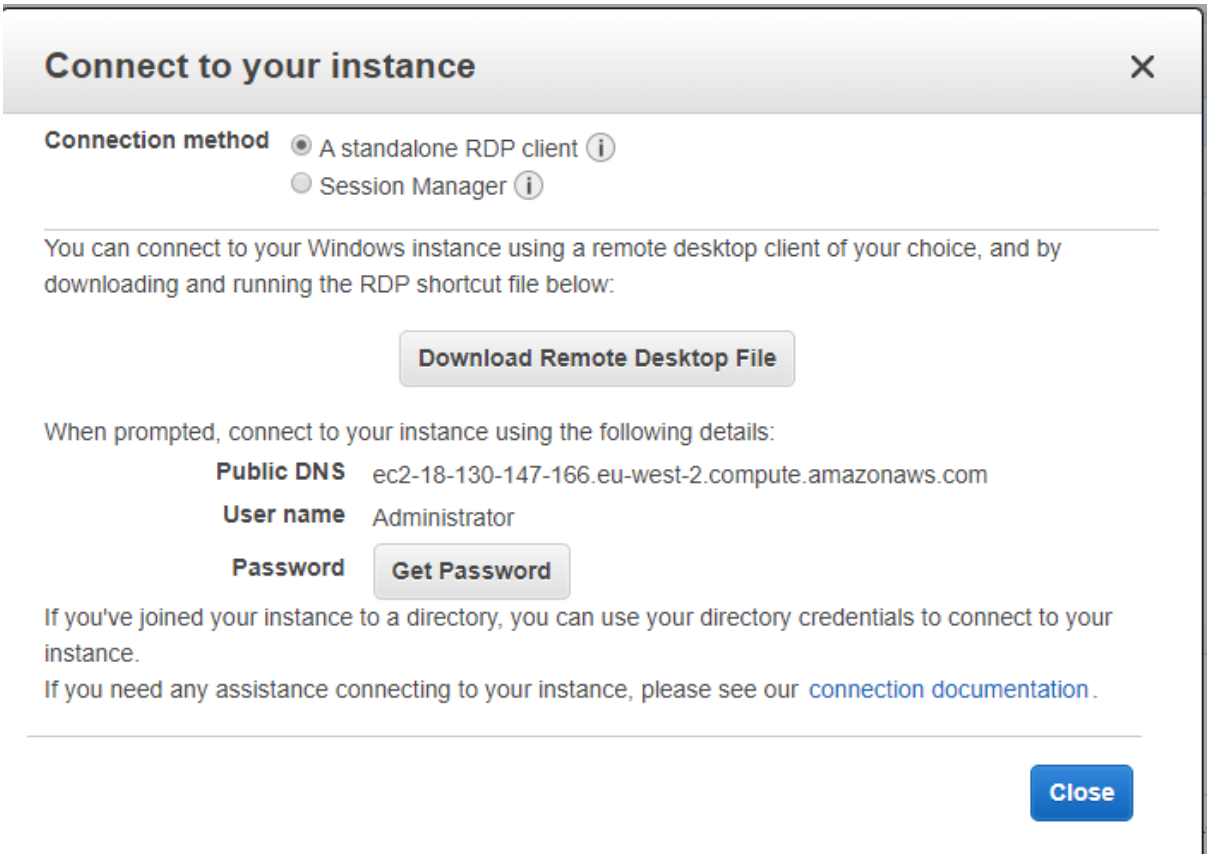
To launch the server, you will be asked to either use an existing key pair or you can create new ones. These are different from the key pairs generated earlier and relate specifically to EC2 instances. Using these allows you to access the instance via remote desktop (RDP). When a key pair file is generated keep it in a known place so it can be accessed when required.



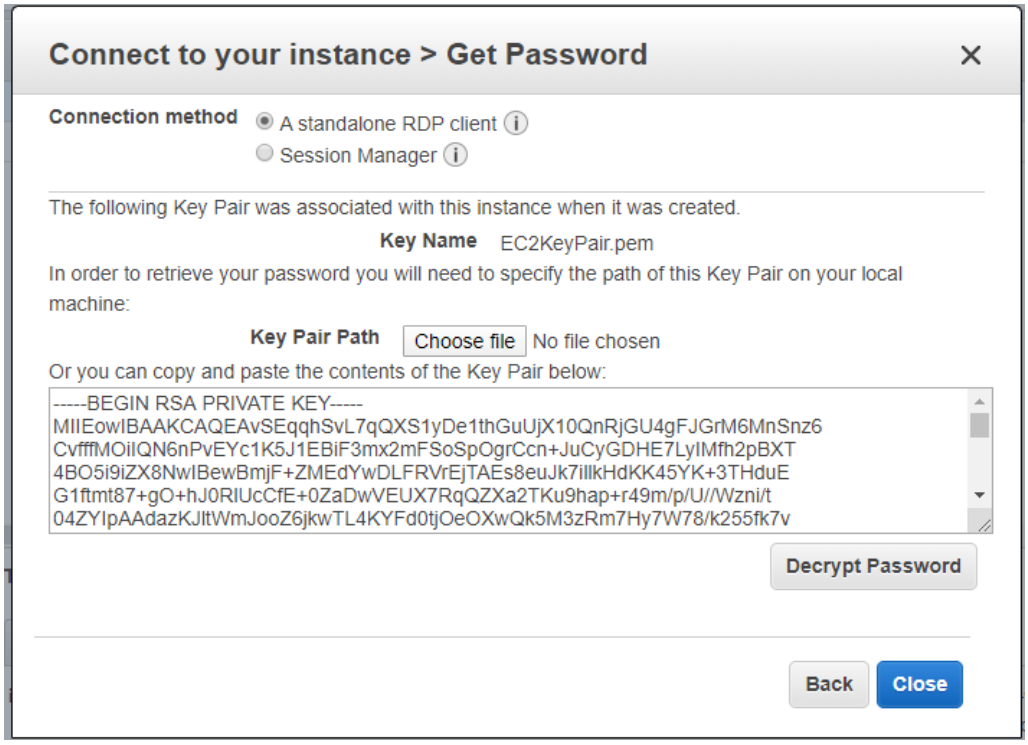
Check the acknowledgement box and launch the instance. The instance will now start and be displayed in the Instances list. The next thing to do is to give it a name and the field can be edited directly in the **Name** field.



To connect to the server from your local machine click on the **Connect** button with the server selected in the list. A dialog box will present 2 options: **Download Remote Desktop File** and **Get Password**.

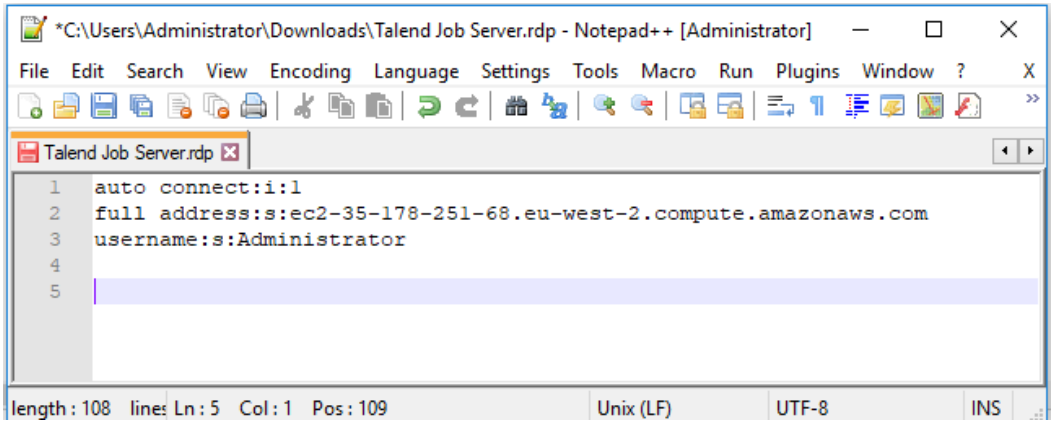


Download the RDP file first and save in a suitable location then click on **Get Password**. Specify the key pair by clicking **Choose file.**

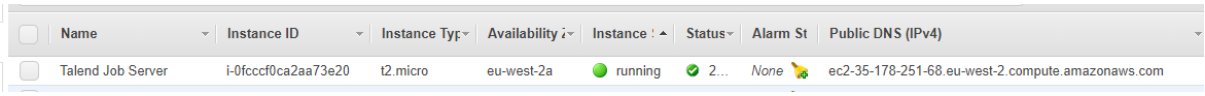


Click on **Decrypt Password** and you will return to the previous screen but with the password now visible. Store this password in a safe place as it will be needed any time you RDP onto this server.

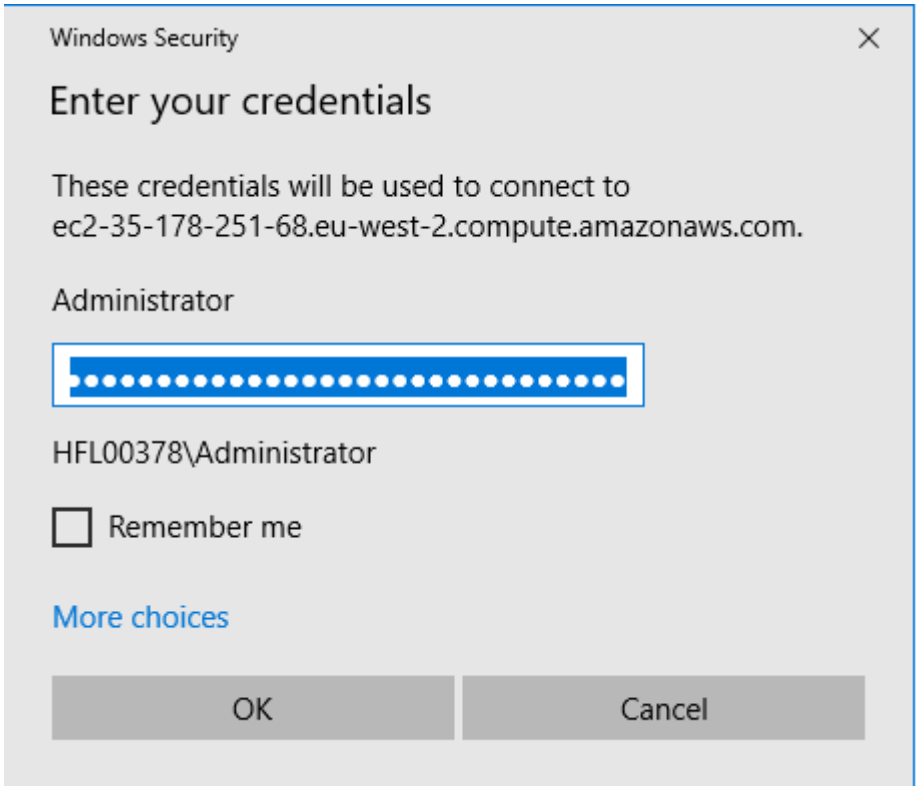
Looking at the RDP file generated for the instance, full address and username are specified.



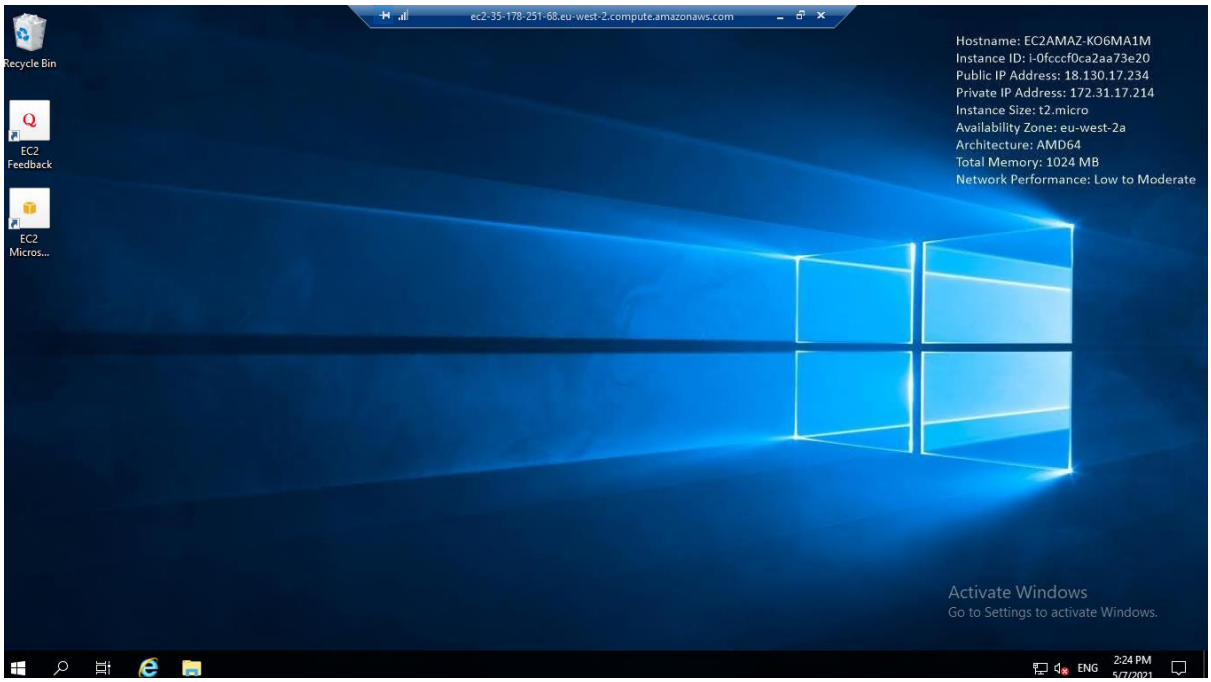
The address matches the public IPV4 DNS shown in the AWS console.



To connect to the server, double-click the RDP file to call the desktop client. Enter the password created and saved previously then press **OK**.



A remote desktop session to the server should now be established and any configurations can be made in the same way as if it was a local pc. The EC2 server is now ready to have Java installed.

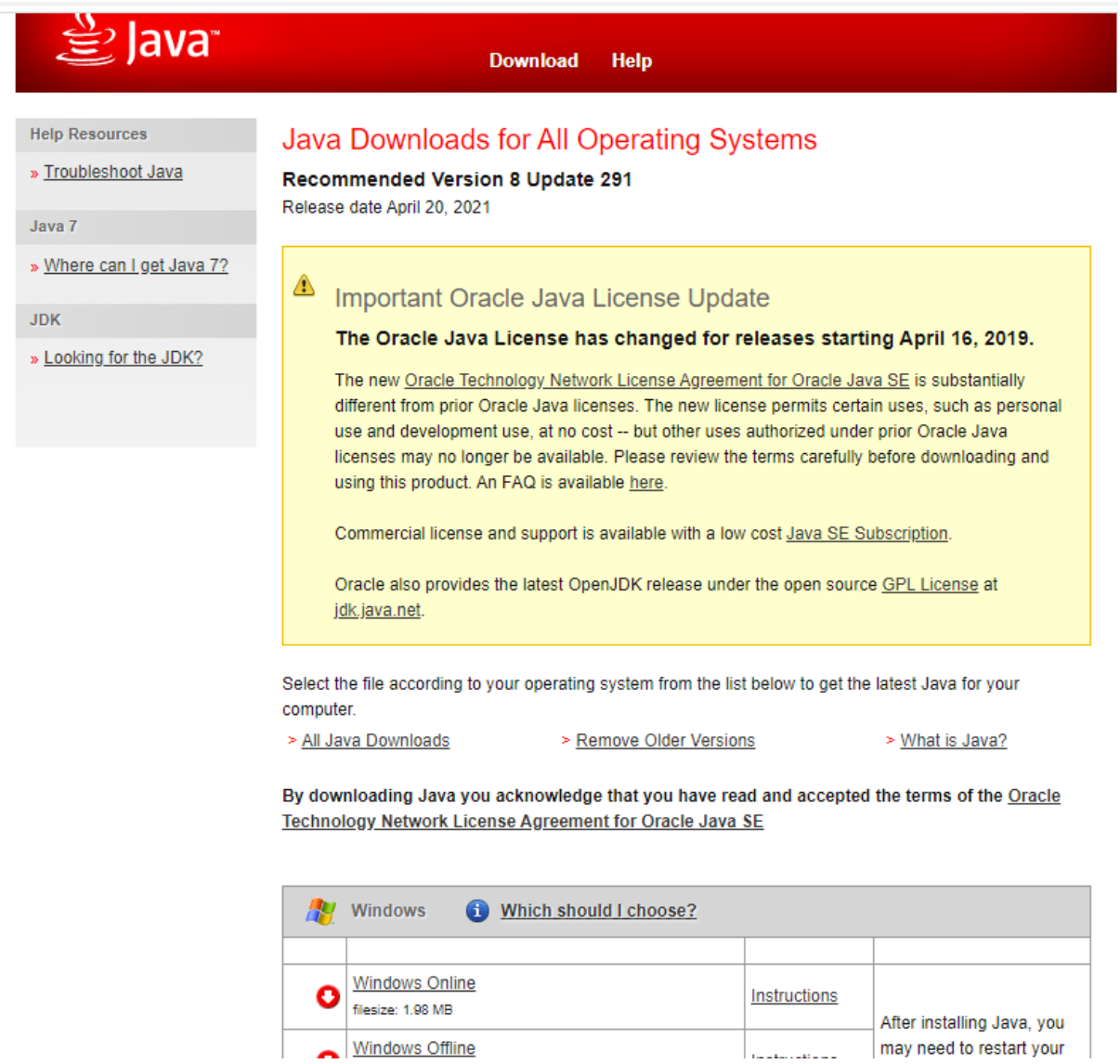


The public IPV4 address only applies to the running instance. If it is stopped and restarted later, a common practice to save resource, the new running instance will have a different address. The old RDP file will no longer create a connection so either create a new RDP file every time an instance is restarted or it is much simpler to just update the DNS in the existing file with the new one from the AWS console. In practice only the four octal byte values will change, the region and availability zone domains remain constant so it may be easier to just change the numbers in the address. The password remains constant for the life of the instance and is not affected by restarts.

7. Install Java 8 JRE onto EC2

For the Talend jobs to run standalone on the EC2 instance it is necessary to install the Java runtime environment (JRE). Talend still recommends version 8 rather than 11.

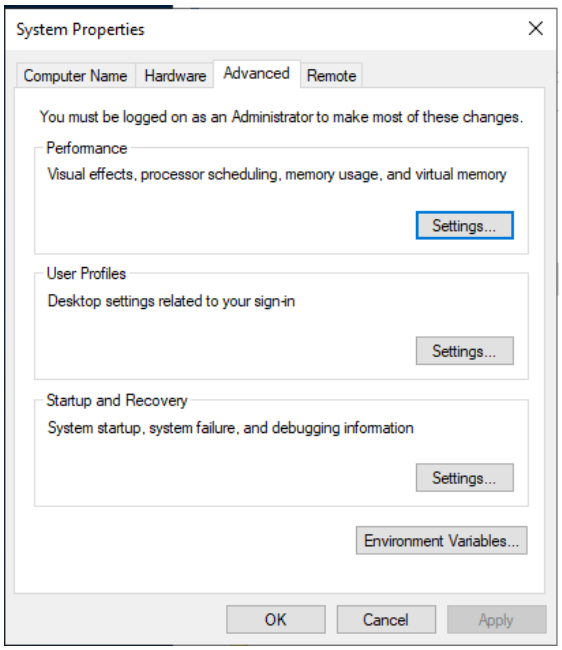
Download the version 8 installer from the Oracle Java site.



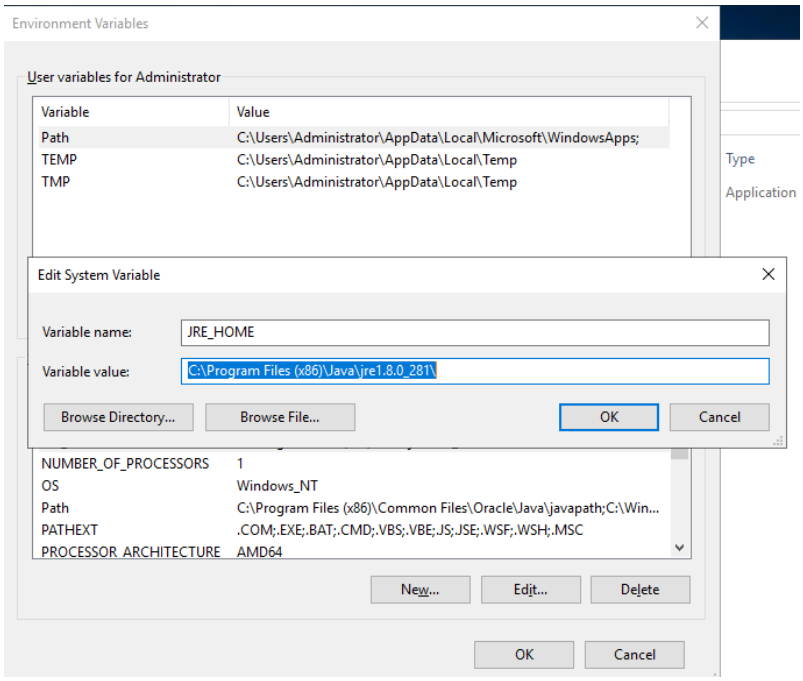
Click on the file to run the installer package and accept the defaults to install Java. This is a straightforward process but in the event of problems there are numerous on-line resources dedicated to the operation, so I won't repeat it here.

Once the installation is complete the next step is to modify the environment variables on the server to allow Talend to run communicate with the JRE correctly.

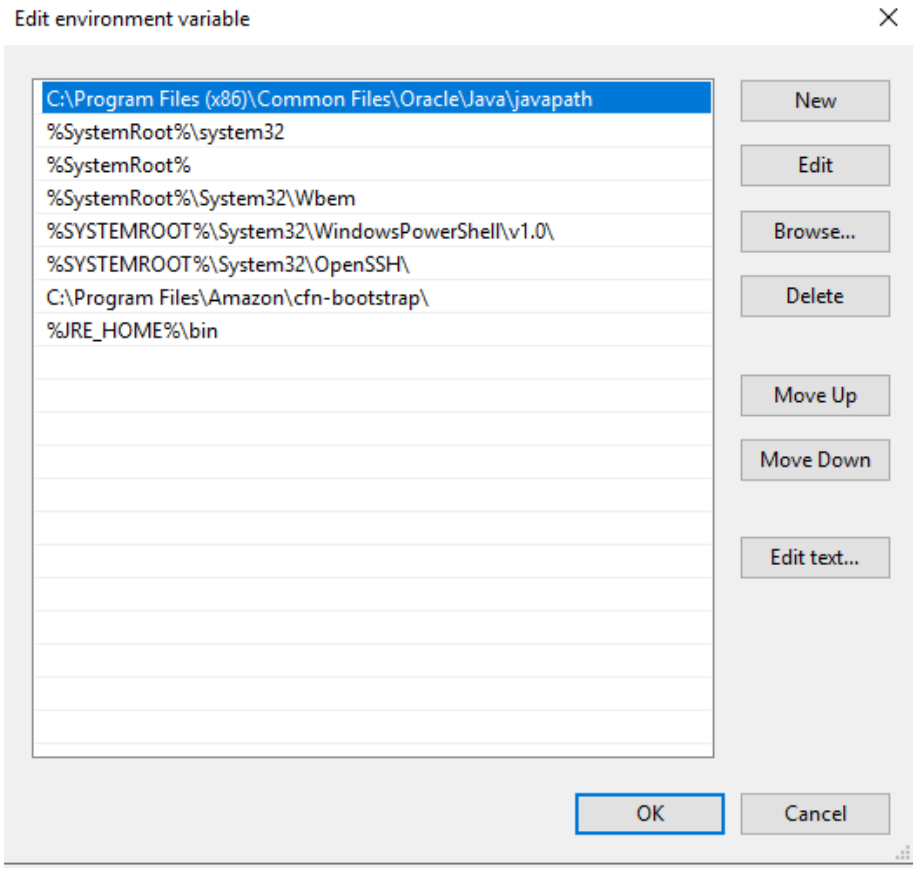
Working in the RDP session for the server, from the **System Properties** dialog click on **Environment Variables**.



In the Environment Variables Dialog click **New** and add a value for JRE_HOME which points to the java location. This should be C:\Program Files (x86)\Java\jre1.8.0_281\ if the standard installation defaults were used.



Save the variable by pressing **OK** then click on the **PATH** variable in the System variables section. Add a reference to the JRE_HOME value to the path.

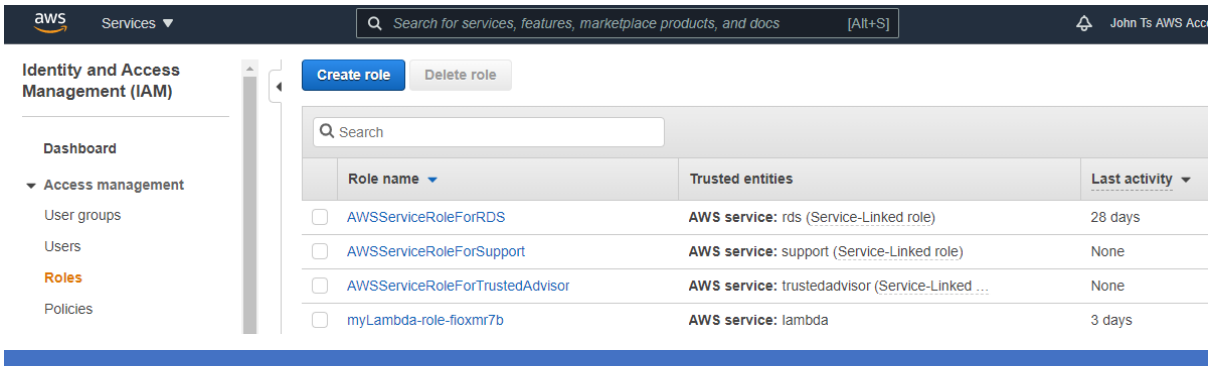


The Java environment is now setup for use by Talend. Notice that only the JRE is required for runtime use, not the JDK. If you wished to install Talend Open Studio (TOS) on the server then the JDK would be required as well.

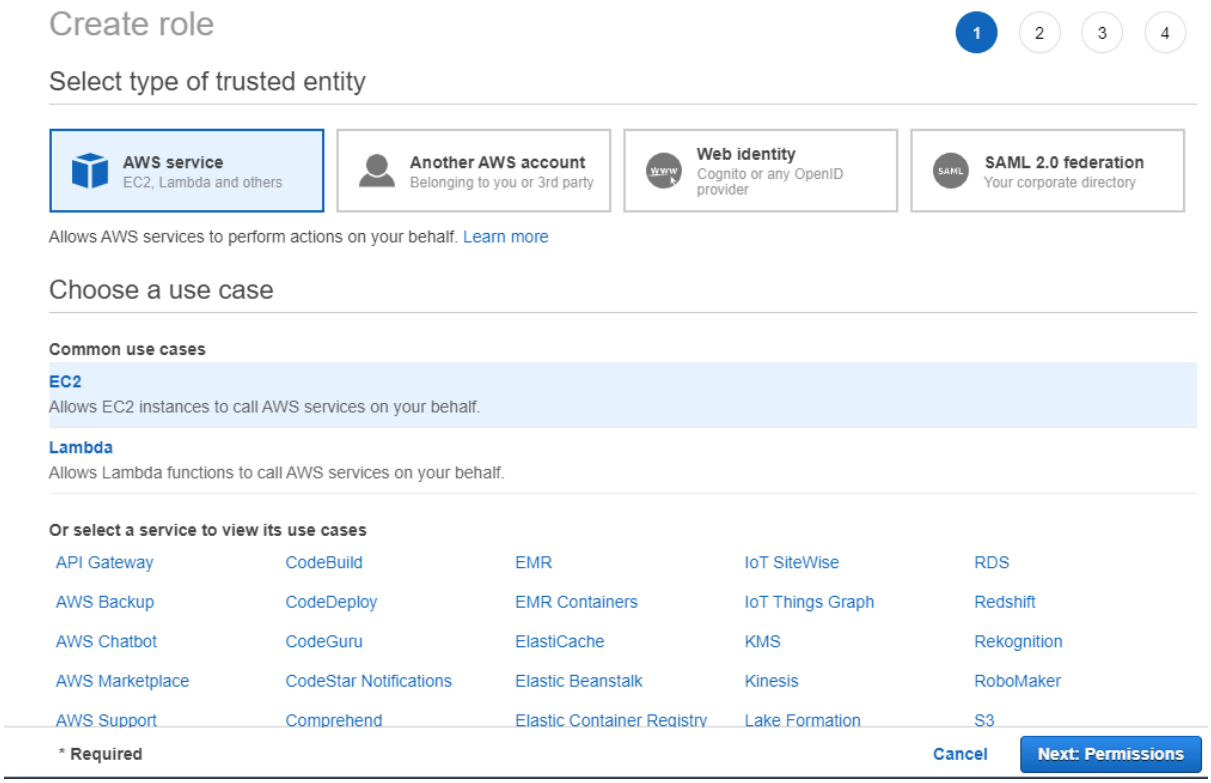
8. Create S3 role for EC2

By default, an EC2 instance has no access to S3 so any jobs running on that server the need to access a bucket will fail. To fix this a role implementing an S3 access policy can be assigned to the server. In normal operations this role should be restricted to the minimum required access but for the proof of concept a simple generalised policy will be used.

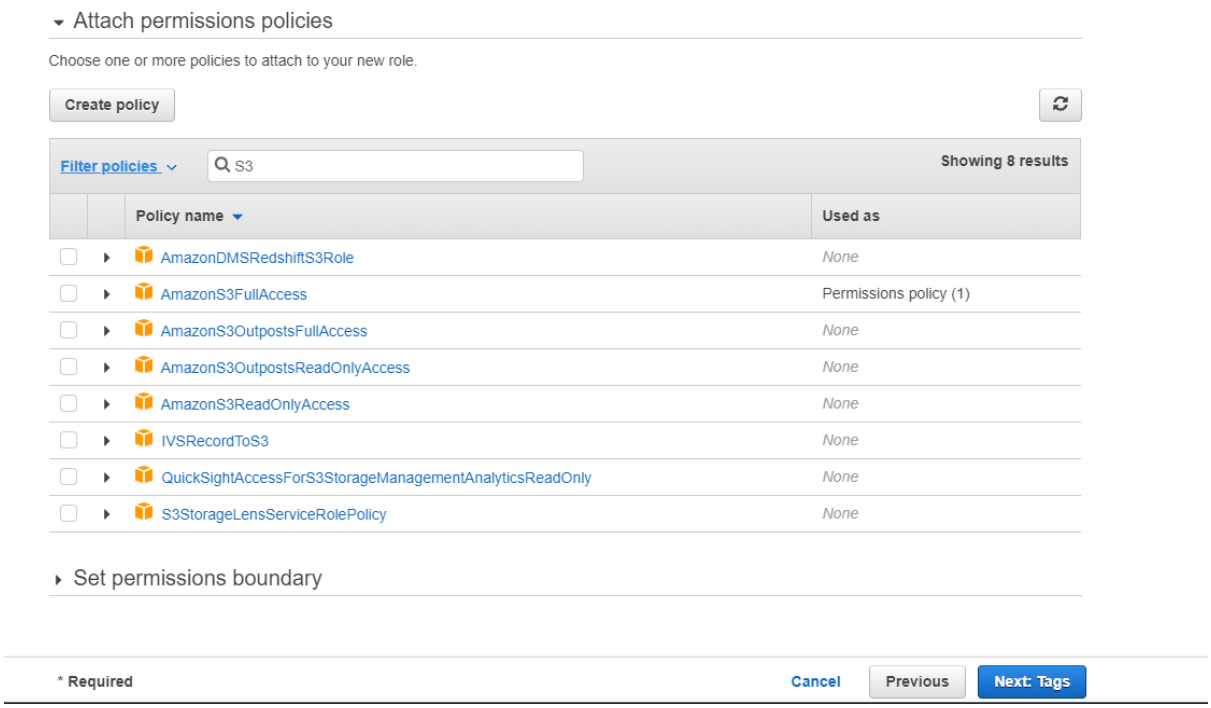
Working in the AWS IAM console select **Roles** from the right-hand pane and click **Create role**



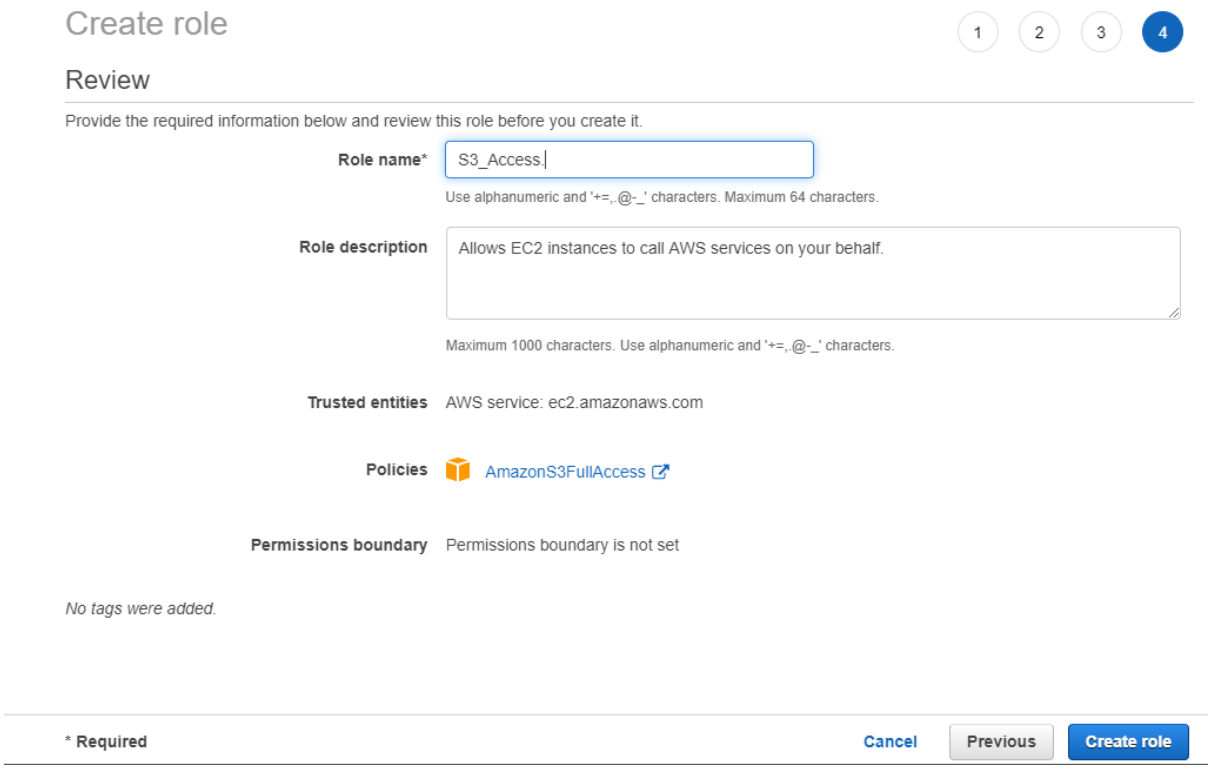
In the Create role screen select AWS Service then EC2 from common use cases. Press **Next:Permissions** to proceed.



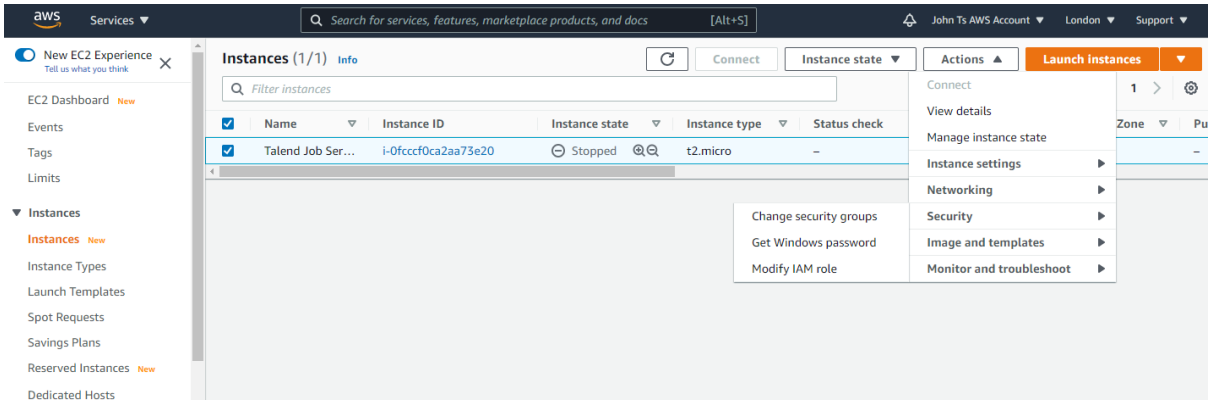
Type **S3** in the Filter policies box then check the box next to AmazonS3FullAccess and click **Next:Tags**



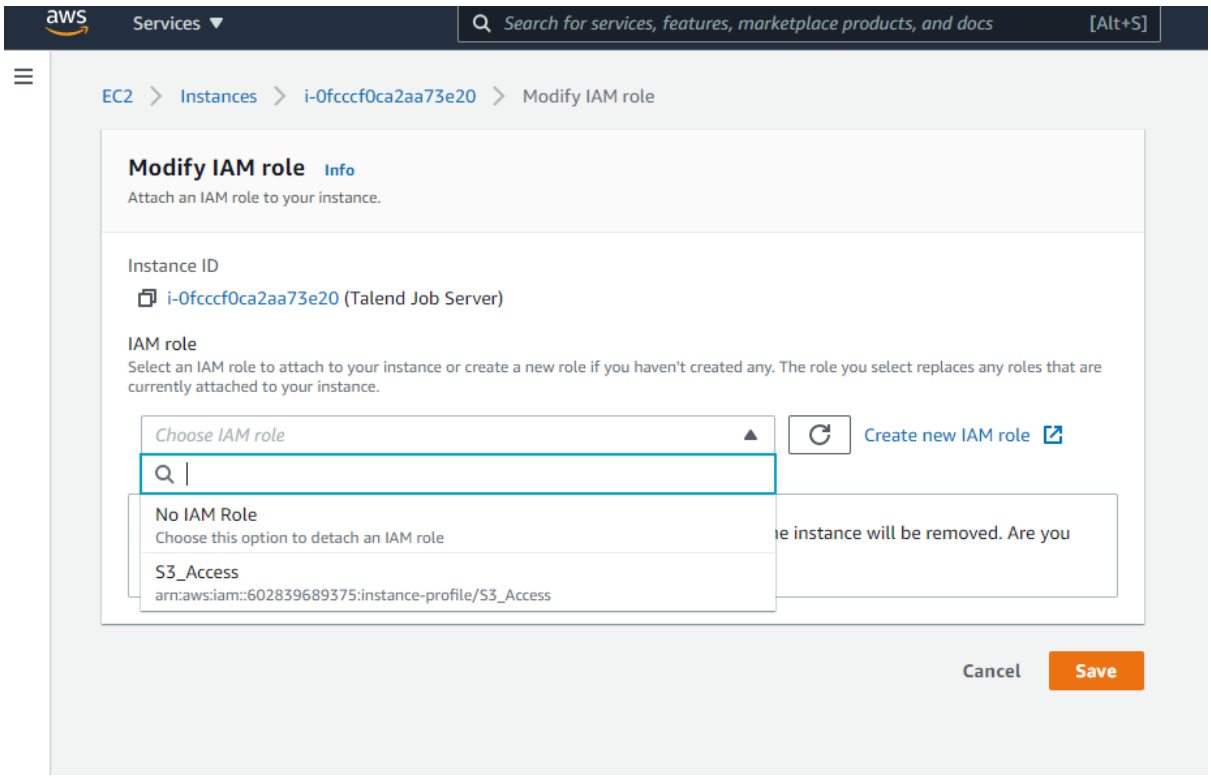
Skip the tags screen and Review the role. Check the policy is shown then give it the name S3_Access and click **Create.**



To attach the role to the EC2 instance, open the AWS EC2 console and select **Instances** from the right-hand pane. Select the instance by clicking the check box in the right-hand column, then click **Actions**, **Security** and **Modify IAM role**.



In the Modify IAM role screen, select the S3_Access role as defined previously and click **Save** to assign the role.



Note only 1 role can be applied to an instance and if other access requirements are necessary, their policies would need to be added to the role.

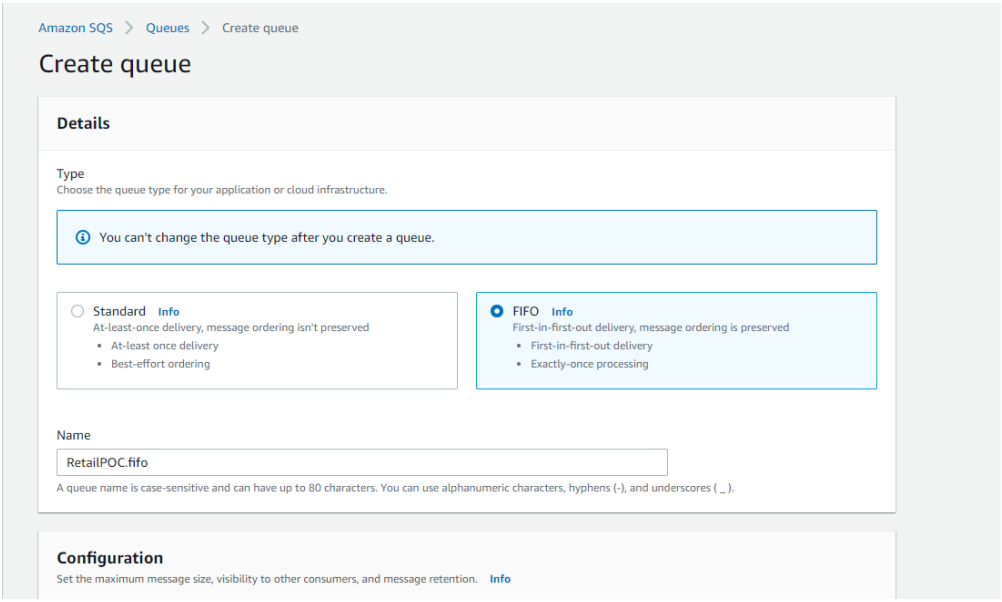
The EC2 instance is now ready for use. Remember that it can be stopped when not in use and restarted as needed to save resource, but it will restart with a new ipv4 address, so the RDP file will have to be changed each time.

9. Create AWS SQS queue

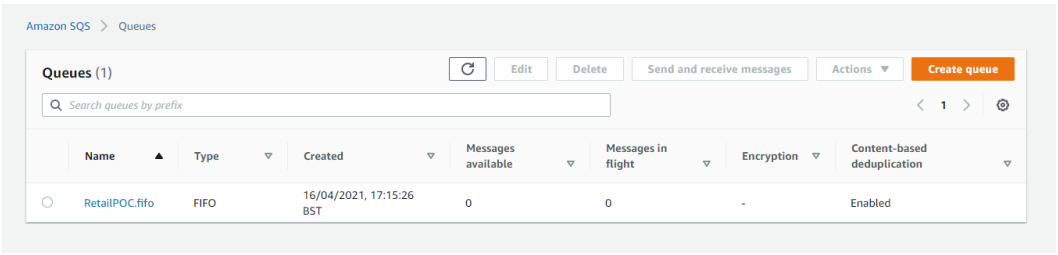
A message queue is a useful way to transfer data between programs. Within Talend, ActiveMQ is often used as it is built into the product, however there are many other queue brokers and AWS has the Simple Queue Service or SQS.

Two types of queue exist: Standard where message ordering is not preserved and FIFO which guarantees first in first out delivery. We will create a FIFO queue for the proof of concept.

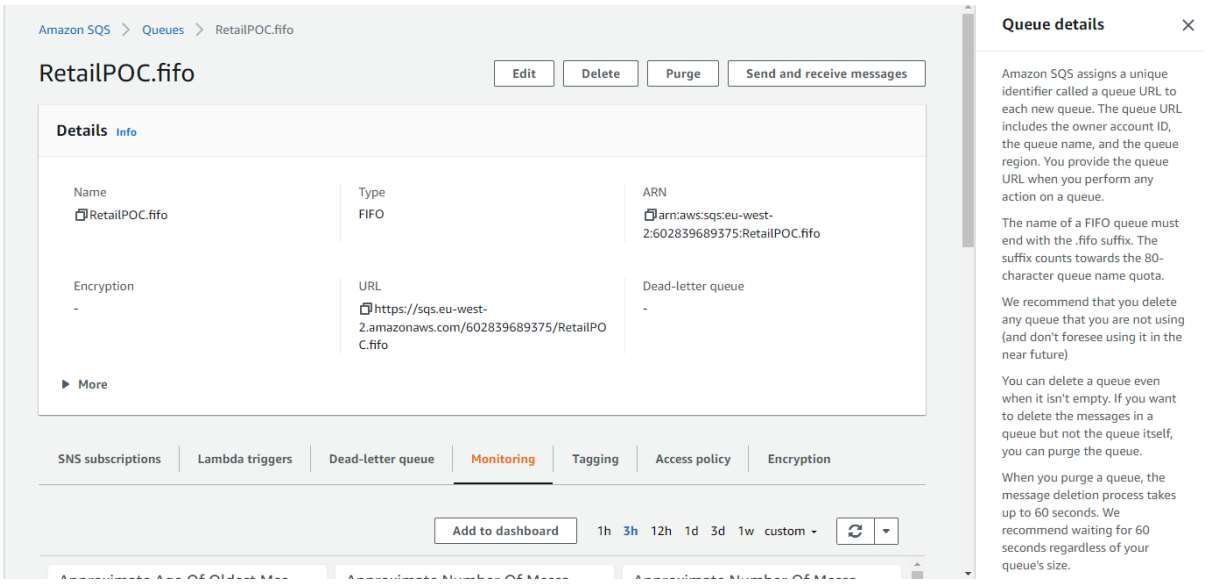
From the AWS management console select **Simple Queue Service** and click **Create Queue**. Check the FIFO radio button and name the queue RetailPOC.fifo. The rest of the config can be left as default for this scenario then click **Create Queue**.



After creation the queue will appear in the list which was also show any current messages available.

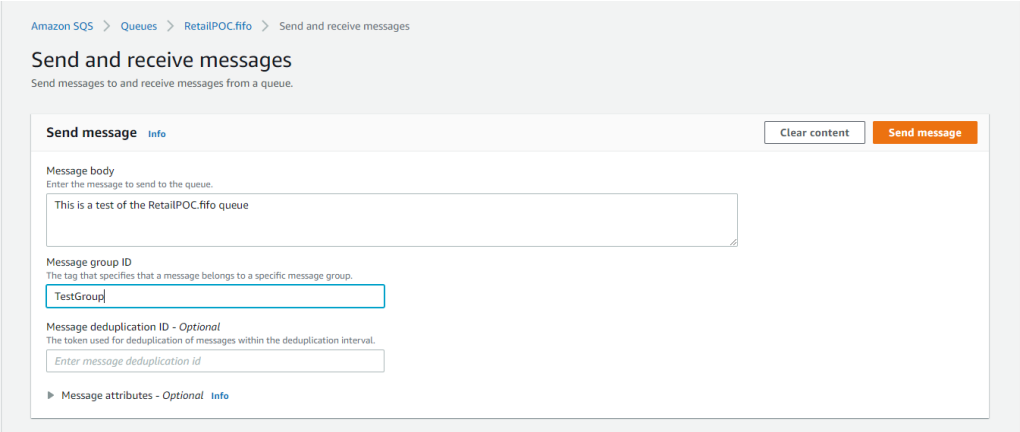


Clicking the queue name will take you to the detail screen which contains full information including monitoring, plus the ability to send and receive messages to test the queue.

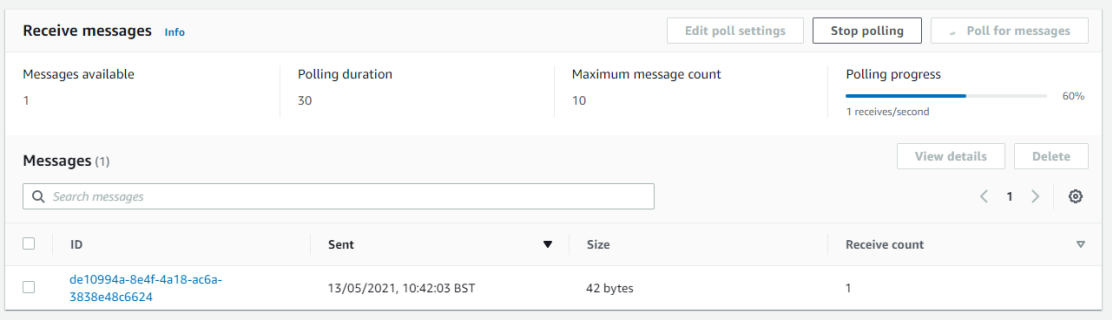


To test the queue, press the **Send and receive messages** option and in the Send Message section add a test message and message group id and click **Send message**.

As this is a FIFO queue it must also have a message group id. FIFO queue logic applies only per message group ID and all messages are sent and received in strict order. Enforcing the group id increases flexibility of the queue allowing it to be serviced by multiple clients using either the same or different group ids and utilising the FIFO capabilities as required. Standard queues do not use this value as it would have no logical purpose and is not displayed in the queue creation screen.



Looking at the Receive message section the message is now shown as available. Click the **Poll for messages** option to retrieve it.

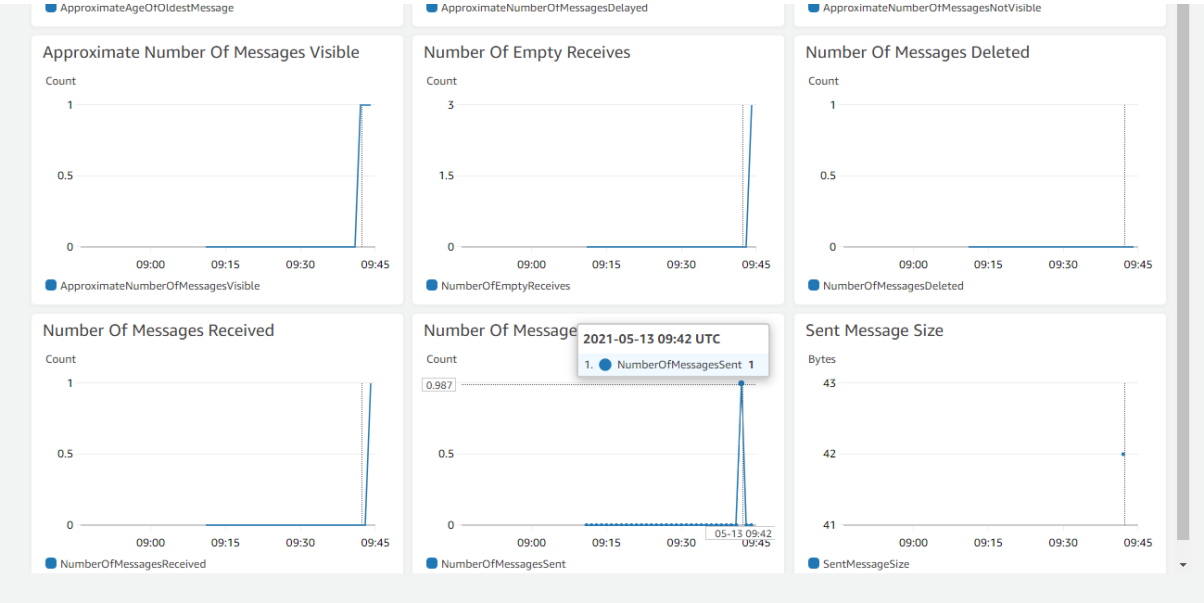


Clicking on the message ID will show the contents.

Return to the detail screen and select the **Monitoring** tab. Set the window to 1 hour and the details of the messages sent and received will be displayed.

This is the default setup, which is fine for the POC, but it can be customised or extended according to requirements.

-



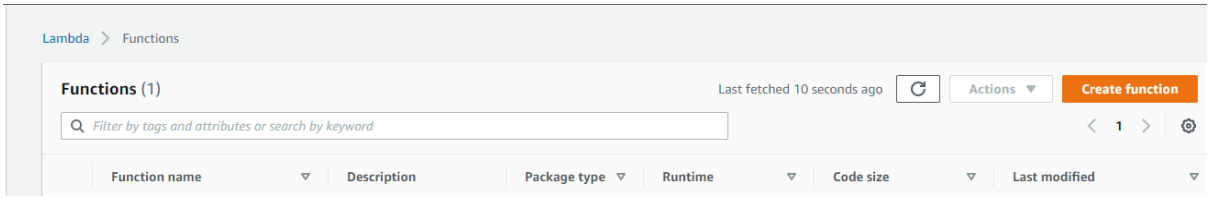
The SQS queue is now ready for use.

10. Create AWS Lambda function

The design goal of the system is for it to respond automatically to the arrival of a file in the S3 bucket and this is achieved by creating a serverless Lambda function triggered by the S3 arrival, which writes the filename to the body of a message stored on the SQS queue defined in the previous section. A Talend job can then monitor the queue and respond to any new messages.

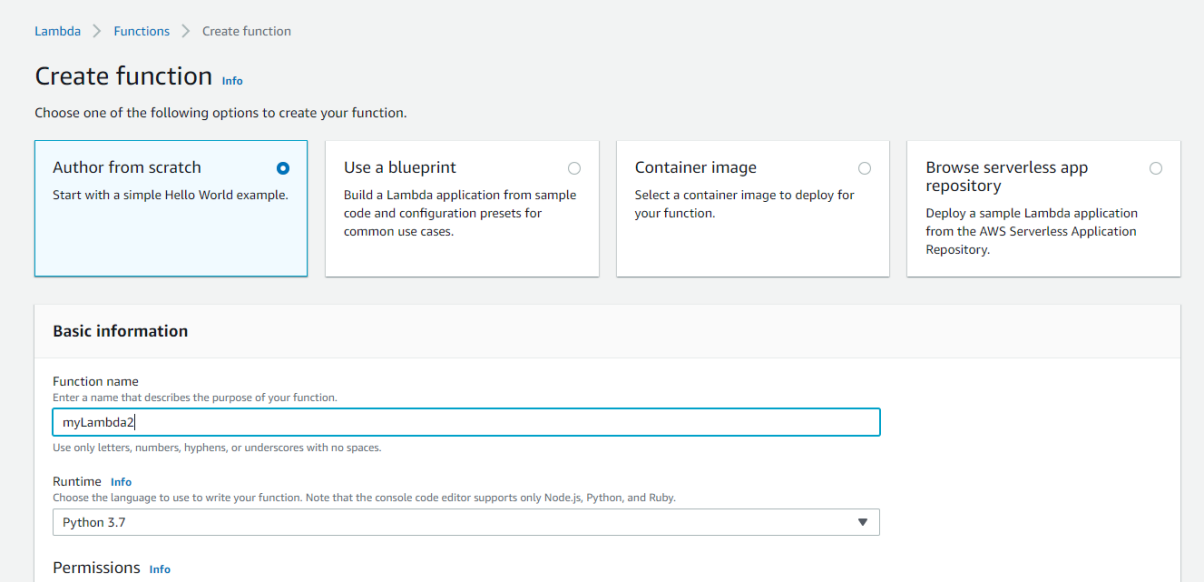
Note Lambda functions are a component of serverless computing meaning that the function code is passed to the service which itself takes care of hosting and execution without any configuration from the user.

From the AWS console select Lambda then click on **Create function**.



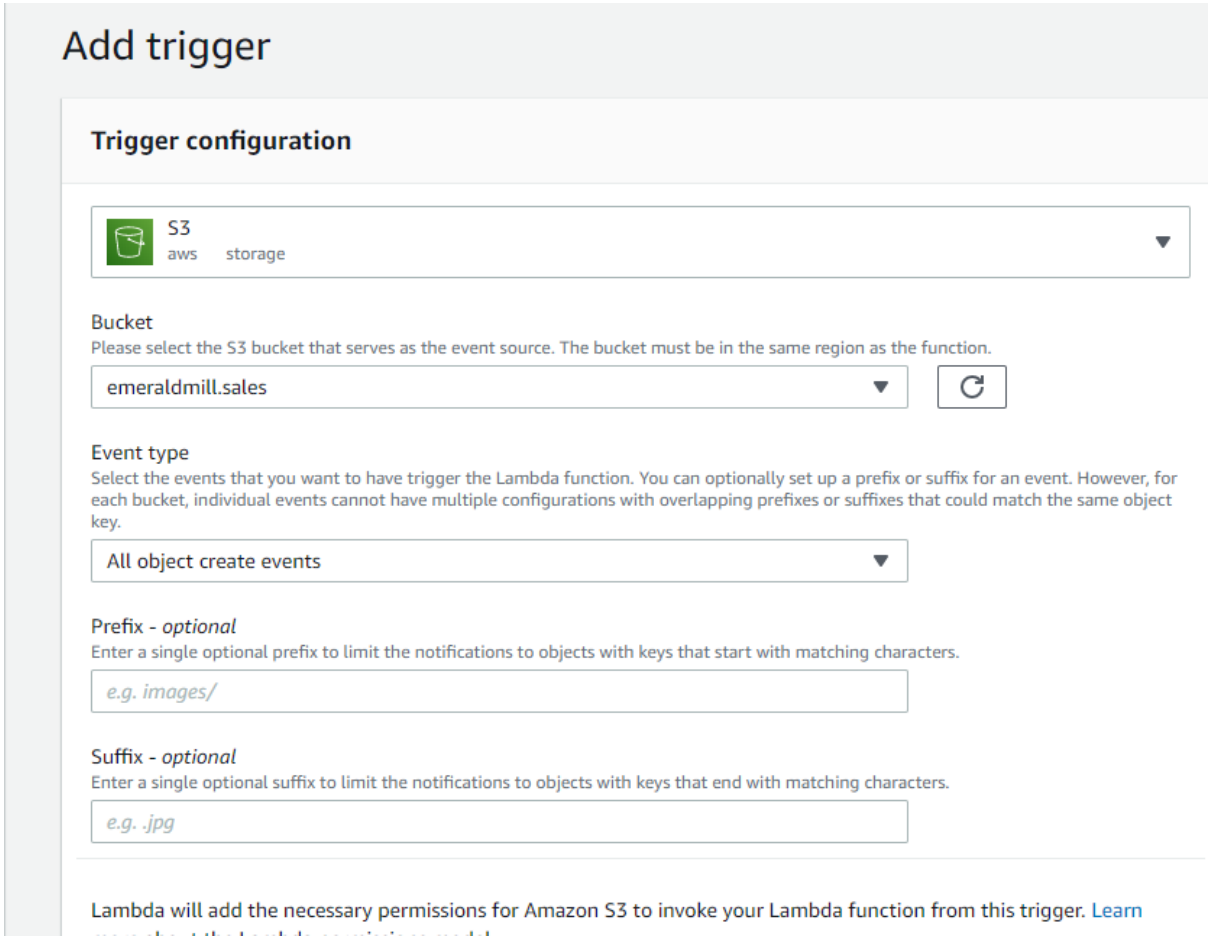
In the Create function screen select **Author from scratch**, name it myLambda and choose Python 3.7 for the runtime. All other defaults are fine in this case so click **Create function**

Lambda offers several choices of language including C#, Java. Node.js, Python and Ruby. Some can be coded directly in the Lambda console whilst others such as C# use external tools such as visual studio. For this simple example Python will be used for clarity.



In the function overview click the **Add trigger** button

Select **S3** from the list and the bucket name from the drop-down options. In this case it will be emeraldmill.sales. Other defaults can be left alone. Click the disclaimer at the bottom warning about recursive functions and press **Add** to create the trigger.

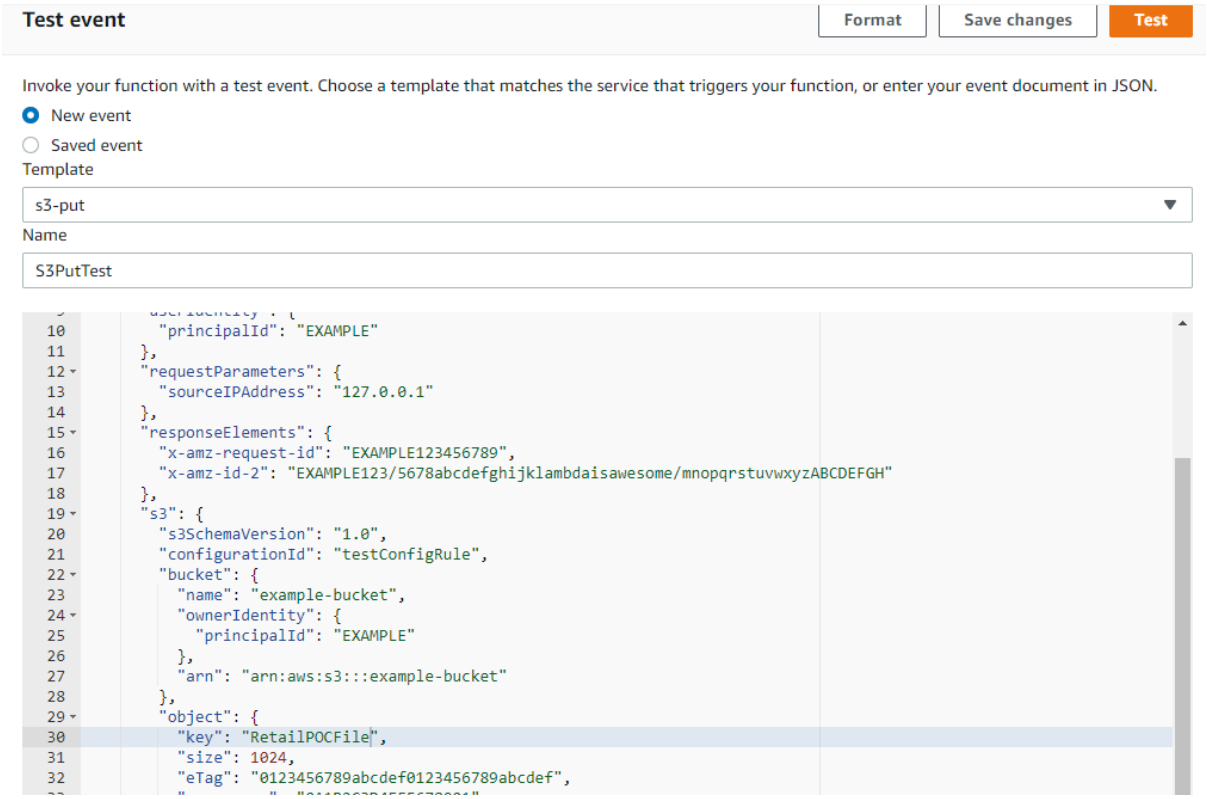


In the code window copy and paste the following code:

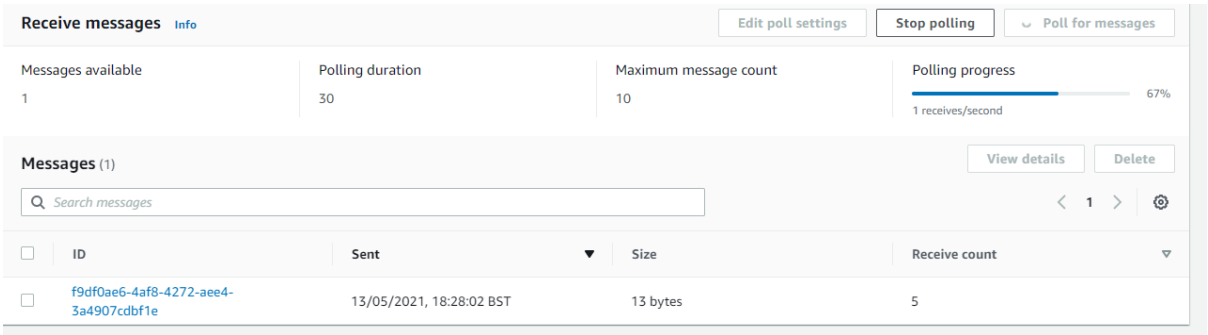
```
"""
Your module description
"""
import boto3
import json

def lambda_handler(event, context):

    sqs = boto3.resource('sqs')
    file_key = event['Records'][0]['s3']['object']['key']

    queue = sqs.get_queue_by_name(QueueName='RetailPOC.fifo')


    response = queue.send_message(
        MessageBody=file_key,
        MessageGroupId='messageGroup1'
    )
```

Click on **Deploy** once complete to deploy the function in Lambda.

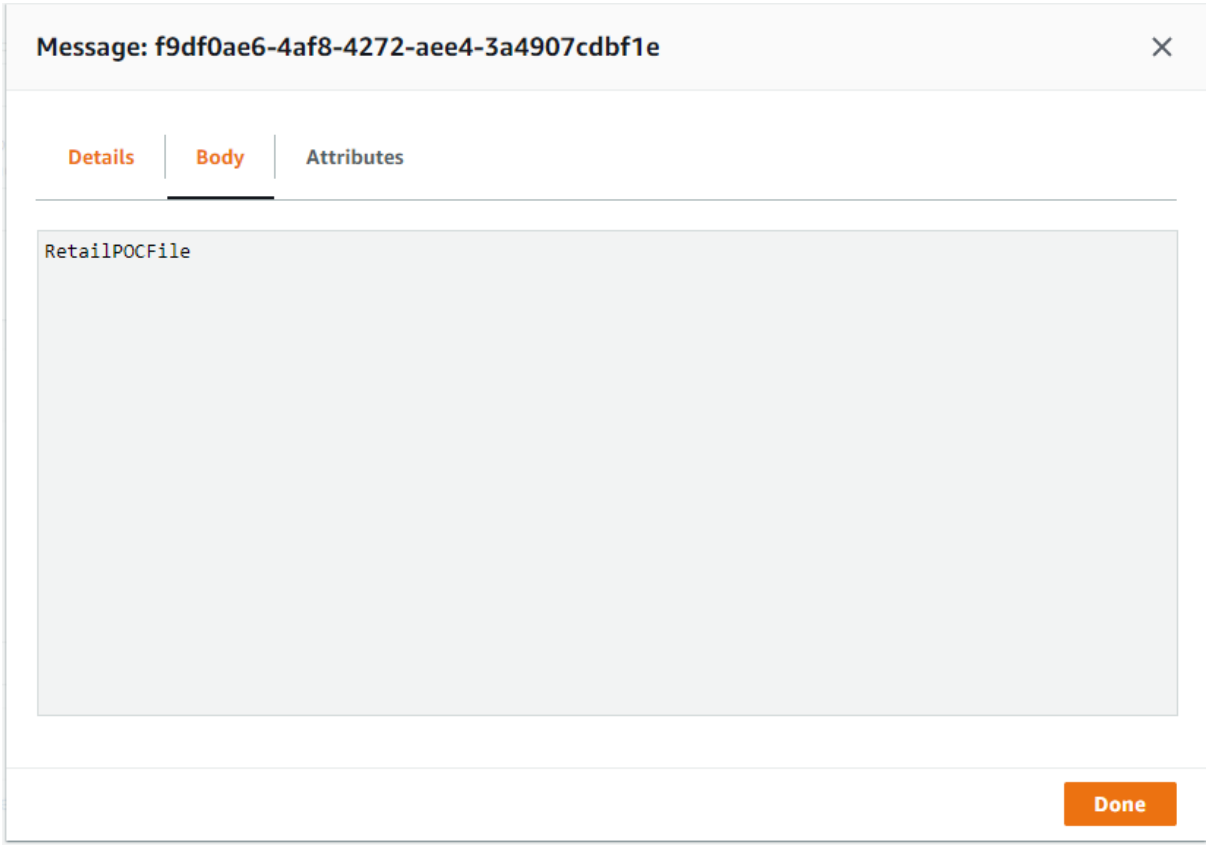
To test the function, select the **test** option from the menu and click **New event** and select **Amazon S3 Put** from the template. Name it S3PutTest and in the "key" element change the value to be "RetailPOCFile". Save the changes and press **Test** to perform the evaluation.



To check the test has worked go the SQS console and select the **RetailPOC.fifo** queue. Click **Send and receive messages** and scroll to the Receive messages section and **Poll for messages**



A message should be shown in the queue. Click on the **ID** to view the message and select the body.



A message showing the key of the S3 object that we modified in the Lambda test is displayed, indicating that the function has responded correctly to the trigger and created the expected SQS message.

Using the test mechanism, we have simulated the S3 event and no file has actually been deposited in the bucket however the AWS template code is designed such that the presence of an actual file would respond in exactly the same way.

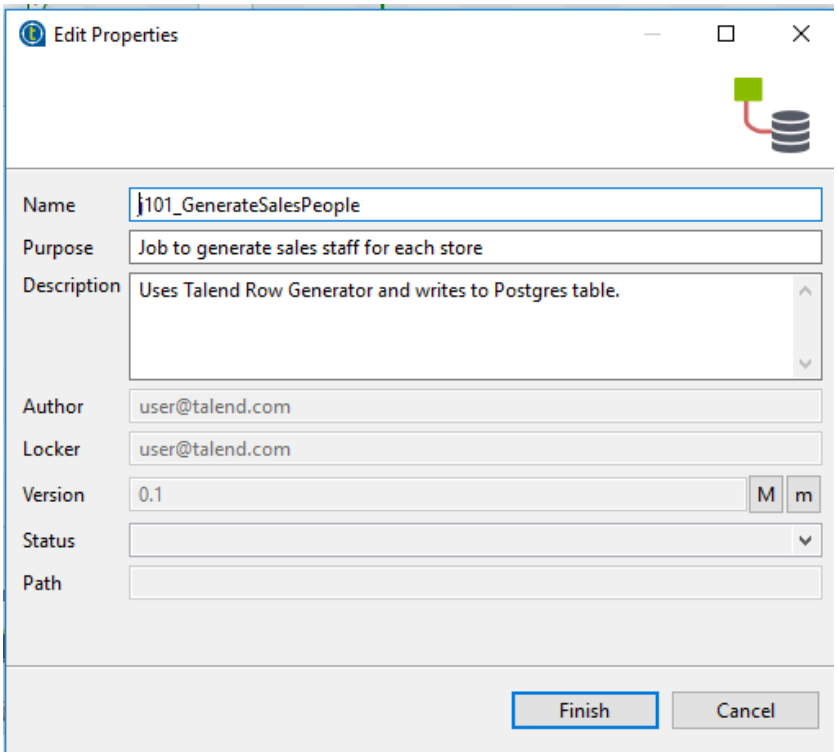
The Lambda function is now complete and will be called anytime a file is loaded into the S3 bucket.

11. Create job to generate store staff

The prerequisites are now complete so we will now look at creating the Talend jobs in Open Studio Big Data Edition. There will be three jobs to create the sales staff records, simulate the work hours and provide a time sheet file and finally to process that file and load the data into SnowflakeDB.

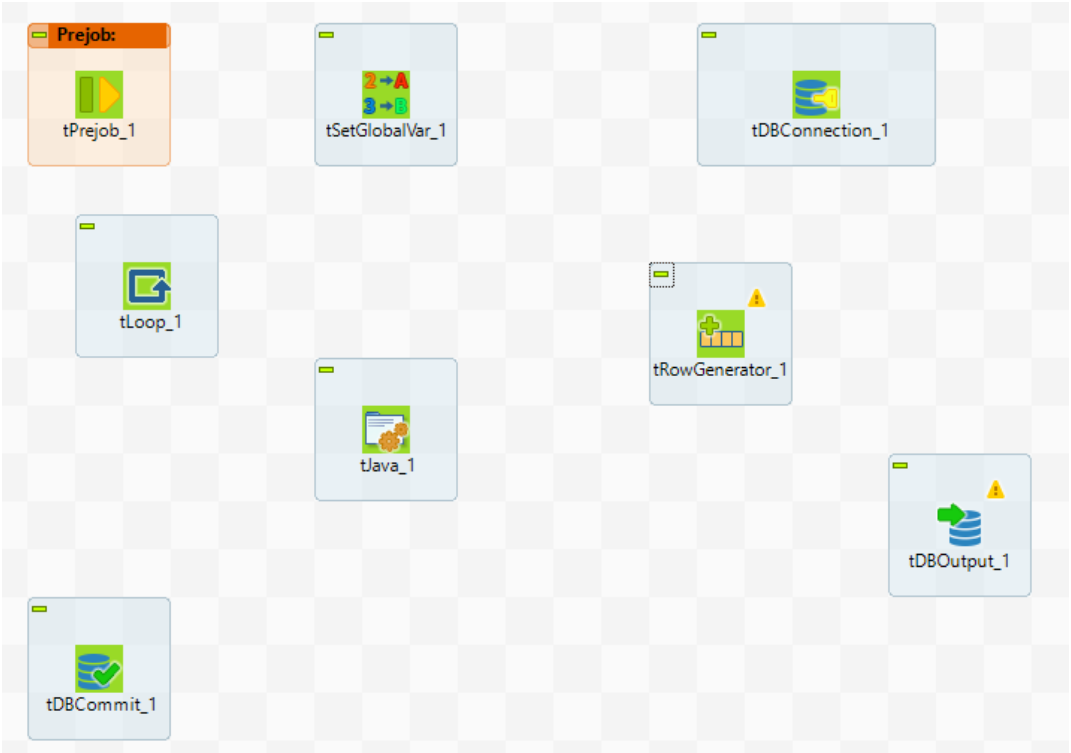
The first job will use the Talend Row generator to simulate sales staff records for each store. The details will be stored in a Postgresql database table.

In the studio repository create a new job and name it j101_GenerateSalesPeople.



Add the following components to the design area:

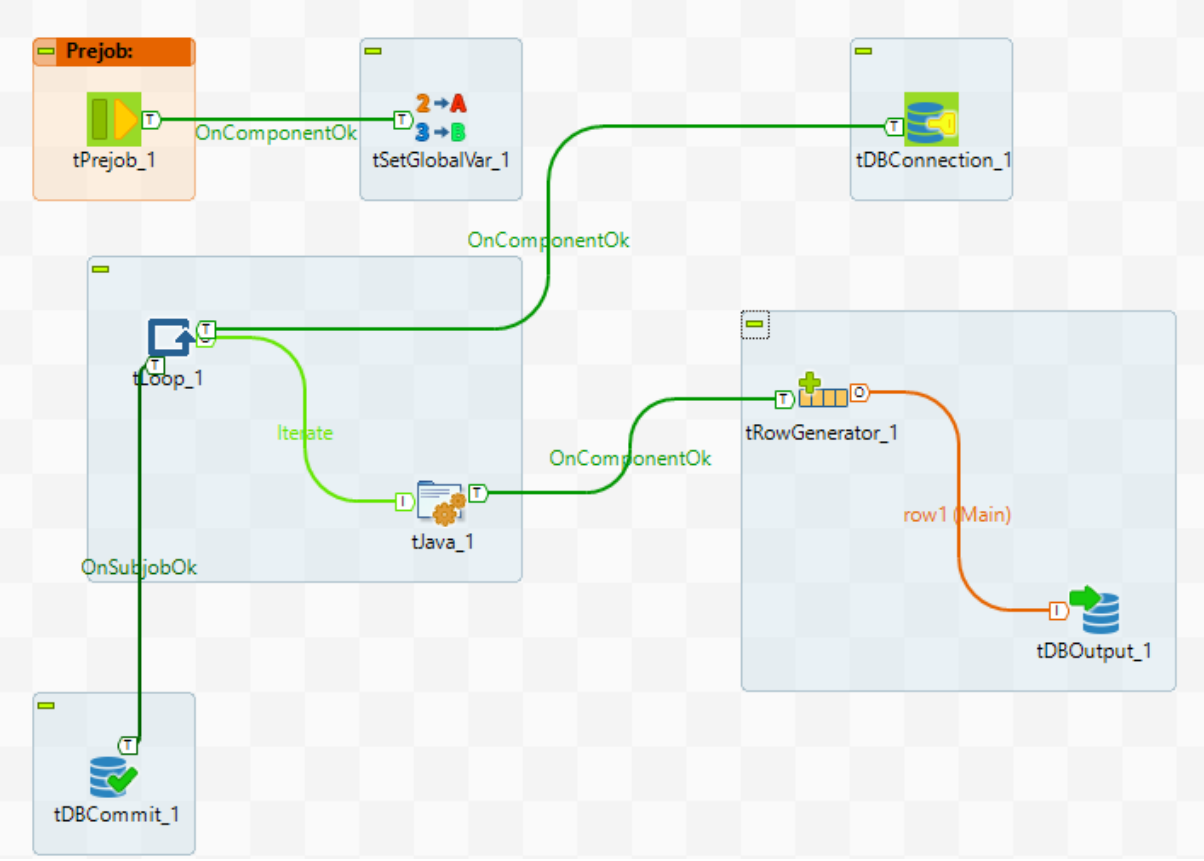
Tprejob	tLoop	tDBOutput
tSetGlobalVar	tJava	tDBCommit
tDBConnection	tRowGenerator	



Join the components as follows:

From Component	To Component	Join Type
tprejob	tSetGlobalVar	OnComponentOk trigger
tDBConnection	tLoop	OnComponentOk trigger
tLoop	tJava	Iterate
tLoop	tDBCommit	OnSubjobOk trigger
tJava	tRowGenerator	OnComponentOk trigger
tRowGenerator	tDBOutput	Row Main

The job should now look like the diagram below:

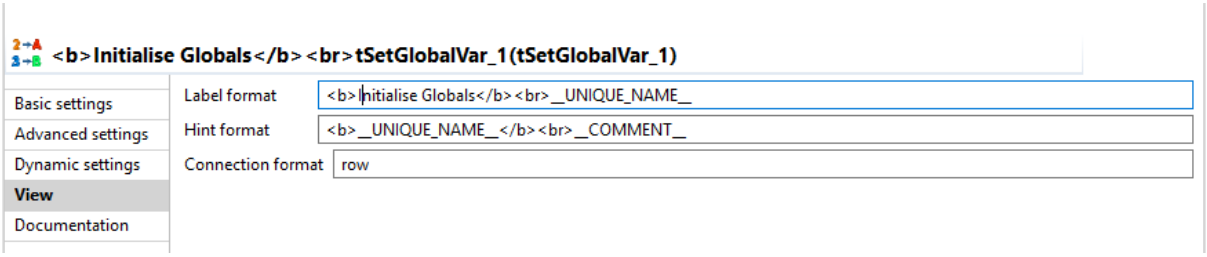


Configure the components as follows:

tPrejob requires no configuration and just initiates a task each time the job starts and is guaranteed to run prior to the main job.

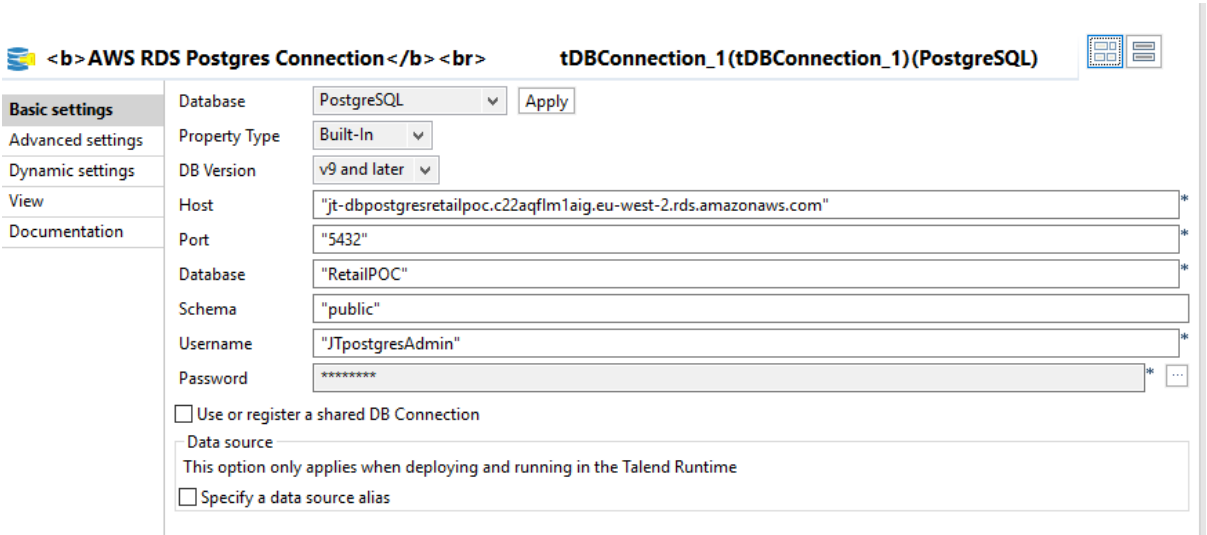
tSetGlobalVar initialises the variable for holding the current store code in the global cache. Click the component in the design area and select the **Component** tab in the bottom pane to show the editor.

From the **Basic Settings** option, press the **green + icon** to create a new row and name it "vStoreCode". Assign a default value of 0 then select the **View** option from the menu.



Add a user-friendly label to component by adding the expression "**Initialise Globals**
" to the start of the Label format field which will show the label in bold together with the actual component name underneath.

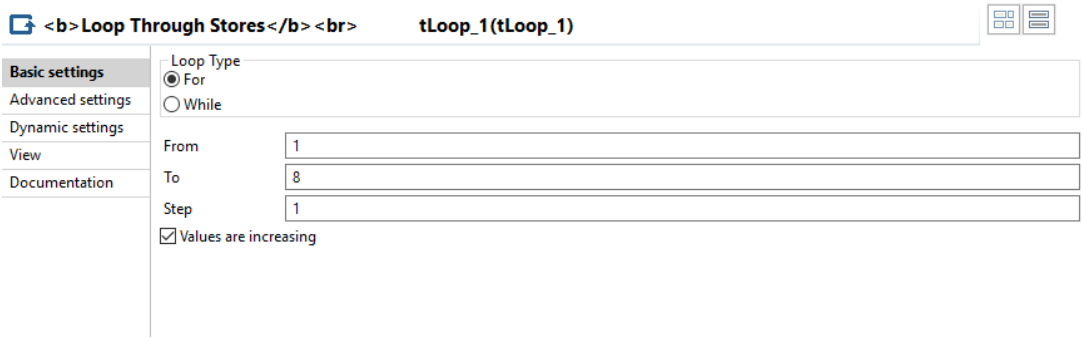
tDBConnection is used to connect to the Postgresql instance which is running in AWS RDS as configured previously.



Select **PostgreSQL** as the database with version v9 or later. The Host value will be the URL shown in the AWS RDS console for the instance. Port should be the standard value 5432 unless a different setup has been used. The Database name will be RetailPOC and the public schema is used. Finally, Username and Password refer to the local user set up in RDS for the instance. A shared connection is not used in this case and since the job will just be ran in Studio a data source alias is not required either and both check boxes can remain unchecked.

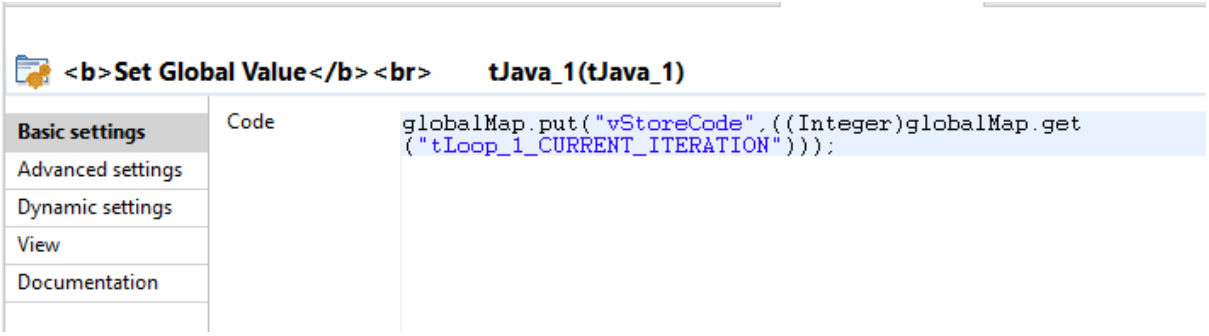
Go into the **view** option and label the component "**AWS RDS Postgres Connection**".

tLoop is used to loop through the 8 stores used in the example and on each iteration run a routine to generate 20 staff records and write them to the database table. On completion of the entire loop the records will be committed, closing the transaction.



As we have a defined number of iterations, a For loop should be chosen going from 1 to 8 in increments of 1. Label the loop "**Loop Through Stores**".

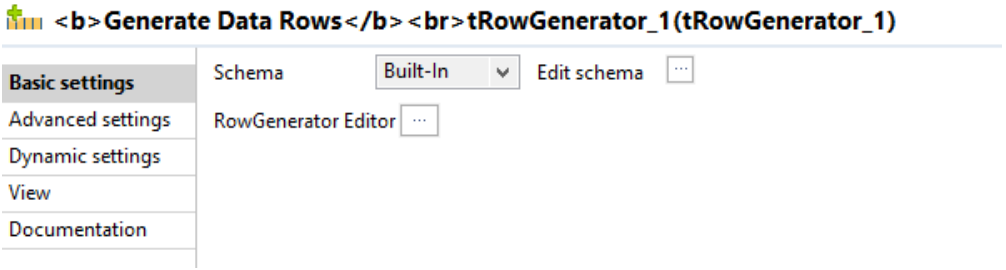
tJava is used to assign the iteration value of tLoop to the global variable vStoreCode. Each call will override the previous value, so it is not necessary to re-initialise the variable each time.



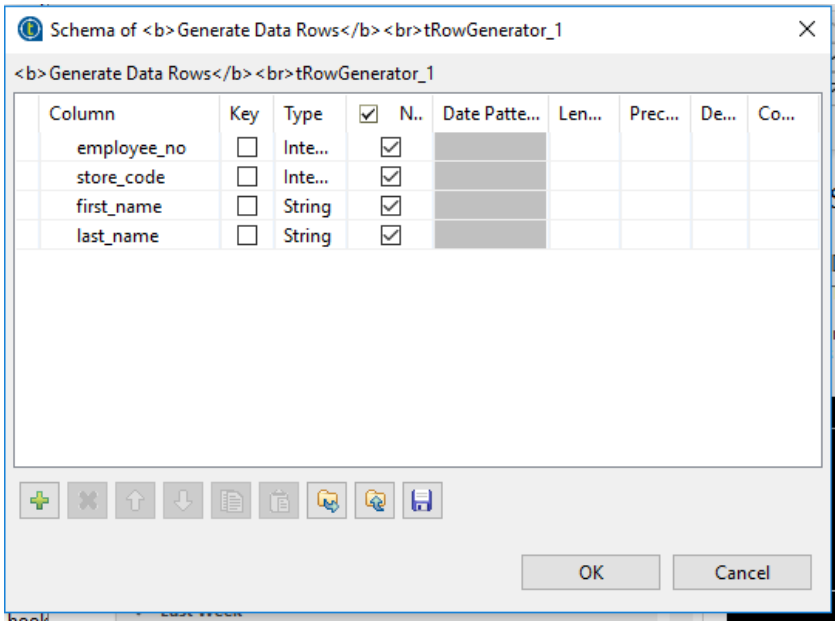
Label the component "**Set Global Value**".

tRowGenerator is a Talend component that allows the generation of Random data in a readable format. For the proof of concept, we will generate some skeleton employee data with an employee number, store code, first name and last name. Store code and employee number will be generated as functions of the loop iteration and the names will be assigned from the row generator.

Note the names will be all male. This is not overt sexism, rather that Talend uses a list of past USA presidents to generate random names and to date these have all been male.



Click the **Edit schema** ellipsis to call the editor.



Click the **green + icon** to add new rows of the following types:

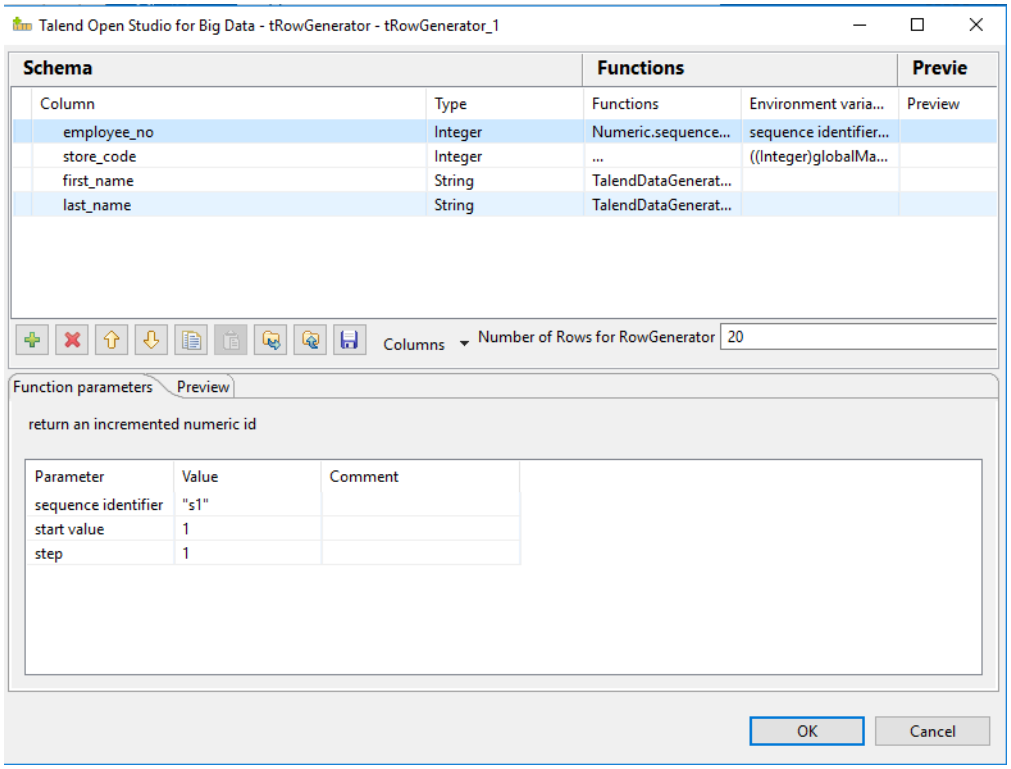
employee_no Integer

store_code Integer

first_name String

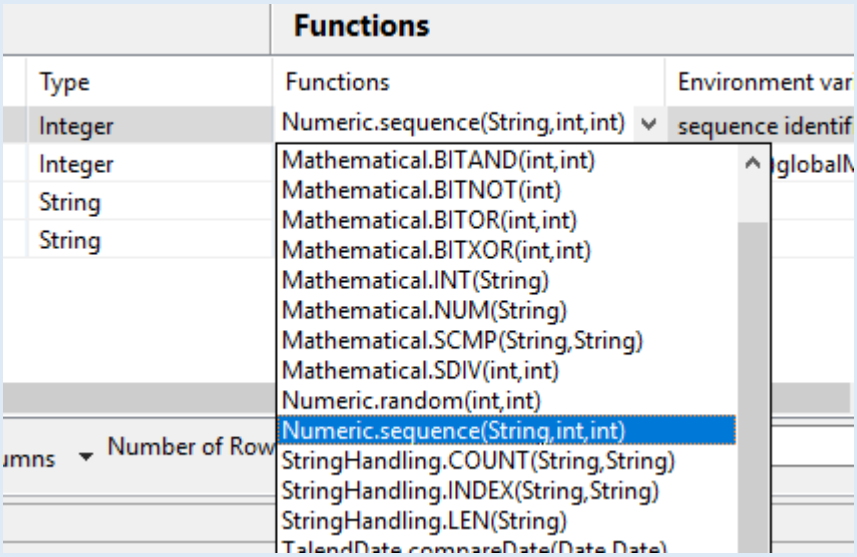
last_name String

Press **OK** to save the schema then click on **RowGenerator Editor**.



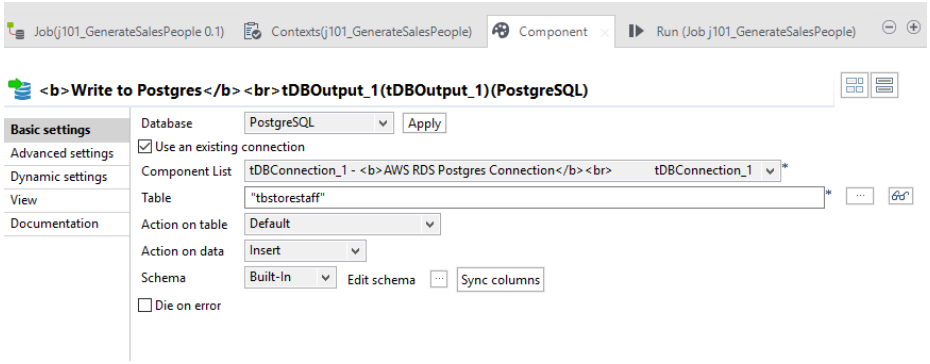
Insert the value 20 for the **number of rows to be generated**.

The schema fields will already be populated, click on each one in turn and perform the configuration steps.

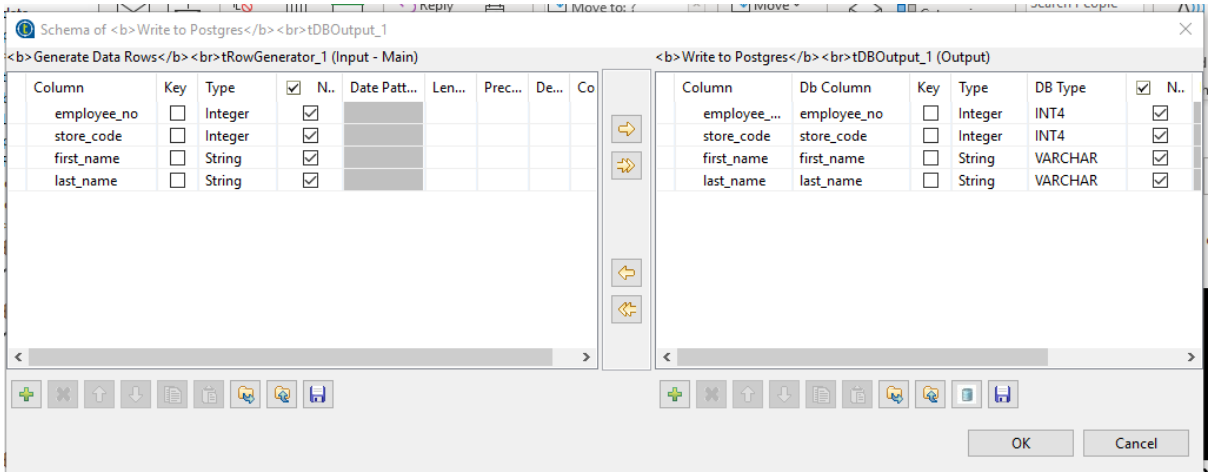
Field	Actions
employee_no	In the Functions drop down list select the Numeric Sequence option  In the Function parameters tab enter the following values: Sequence identifier “s1” Start value 1 Step 1
store_code	Select the ellipsis option from the top of the drop-down function list which indicates a custom value. In the Function parameters tab enter the value ((Integer)globalMap.get("vStoreCode")) to retrieve the current value of the Store Code global variable.
first_name	Select TalendDataGenerator.getFirstName() from the Functions drop-down.
last_name	Select TalendDataGenerator.getLastName() from the Functions drop-down.

Click **OK** to save the configuration then label the component “**Generate Data Rows**”

tDBOutput writes the generated rows to the database table.

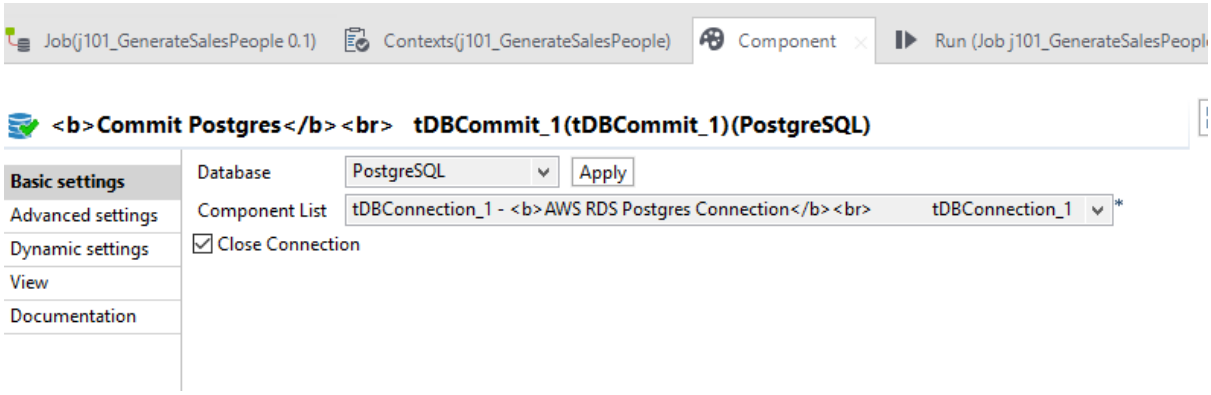


Select **PostgreSQL** as the database type and check the **Use an existing connection** box. In the Component List select the DB connection defined previously. Enter "**tbstorestaff**" for the table and leave the Action on table setting as Default. Action on data should be set to **Insert**, then click **Sync columns** followed by **Edit schema** and verify the schema looks the same as below.



Click **OK** to exit the schema editor then label the component "**Write to Postgres**"

tDBCommit will finally commit the Postgres transactions and close the database connection.

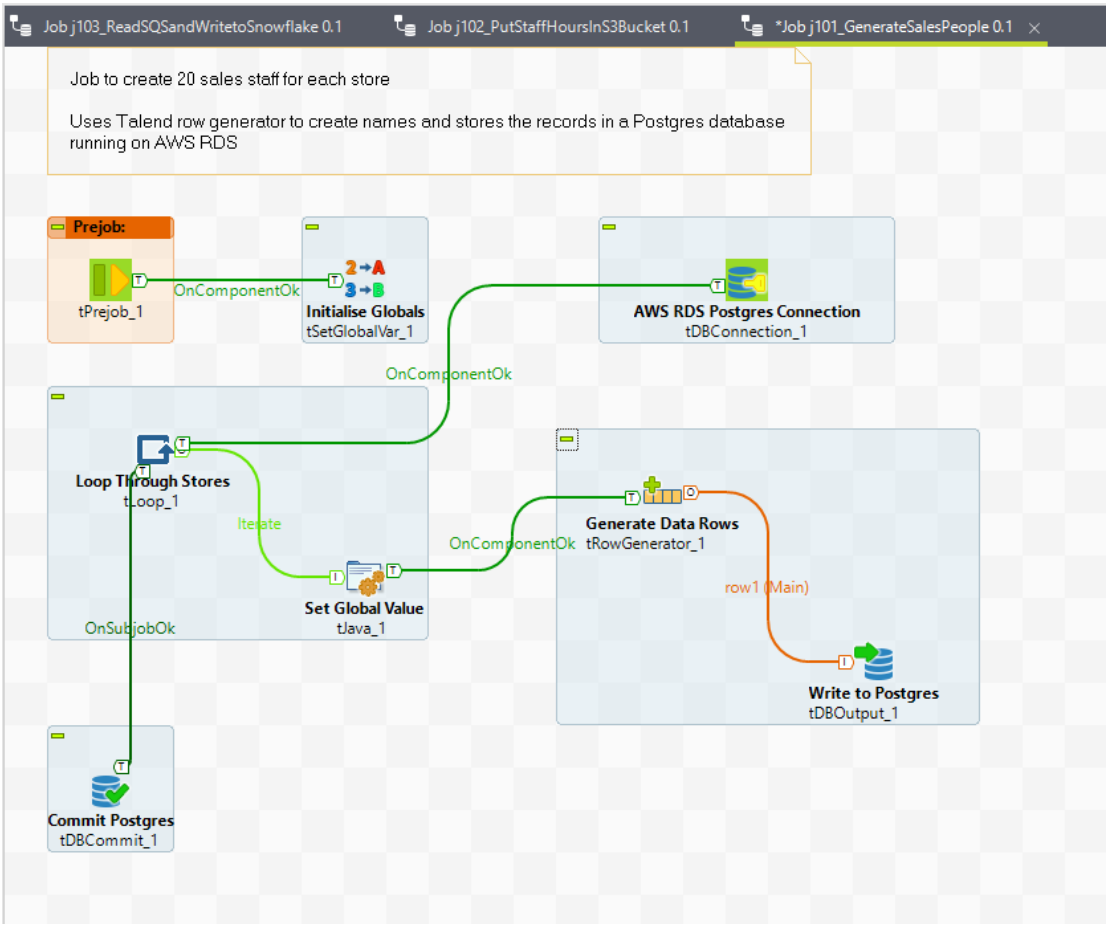


Select **PostgreSQL** as the database type and choose the database connection component from the list. Check the **Close Connection** option and label the component "**Commit Postgres**"

Ensure that the connecting tLoop to tDBCommit is of the type OnSubjobOk in order that it fires after the loop has completed and all data is written. If an OnComponentOK trigger is inadvertently used instead it will call the commit immediately the loop is started and will close the database connection before data had been written, leading to an error when tDBOutput tries to access it.

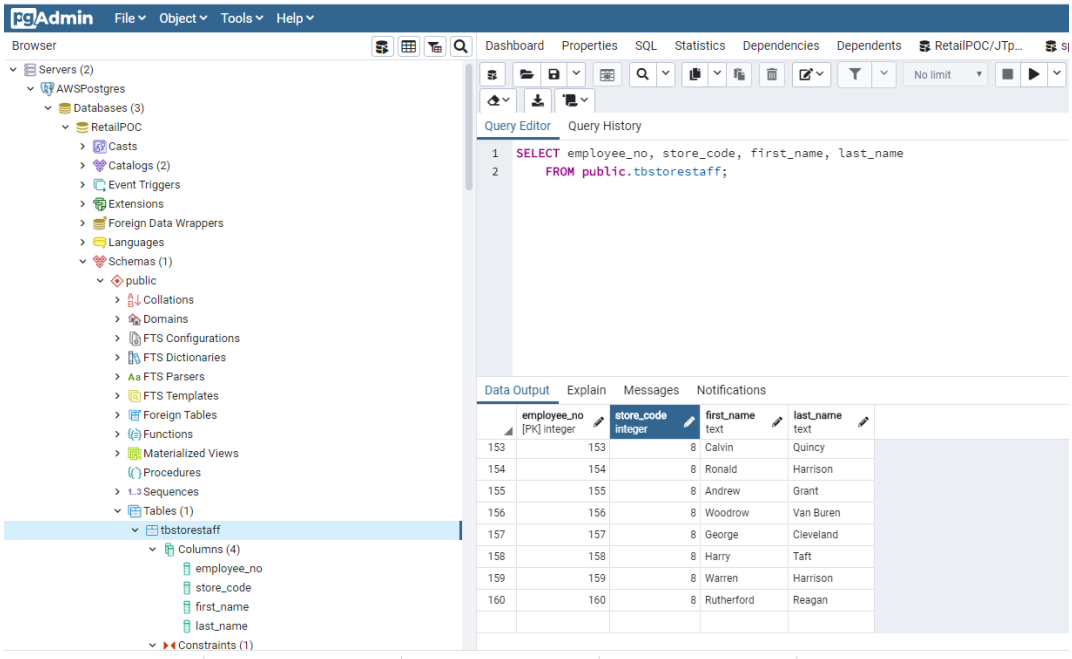
Complete the job by adding a note to the screen similar to:

The completed job should now look like this.



The job is now ready to run in Talend Studio but check that the database table is empty prior to execution or a primary key violation may occur.

After completion the results can be viewed in PGAdmin or any similar client.



This is a rudimentary routine designed to demonstrate both the row generator and loading data into PostgreSQL hosted on AWS RDS. Flexibility could be enhanced by using context variables loaded via file or database to vary parameters such as number of store staff and total stores and update or insert used to prevent key errors on subsequent runs.

12. Create job to generate timesheets

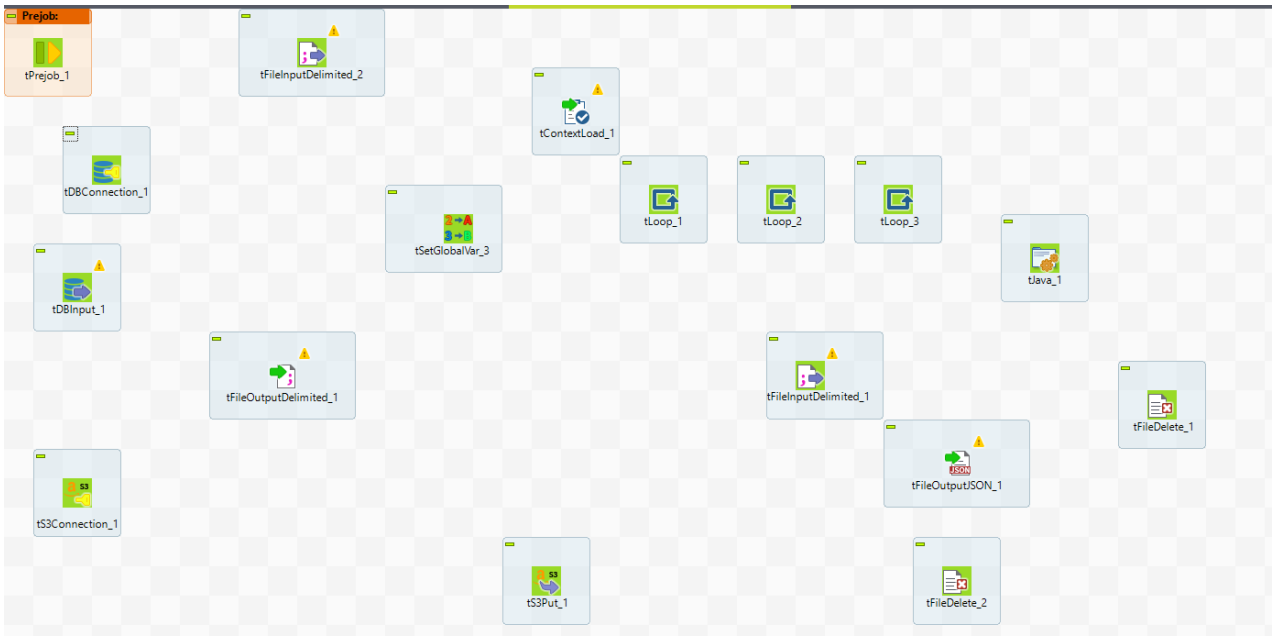
The second job to simulate the creation of timesheets for the staff generated in the previous job and write the data as a JSON file to the S3 bucket defined earlier. This will trigger the AWS Lambda function which writes a message to AWS Simple Queue Service (SQS).

Create a new job in the repository called j102_PutStaffHoursInS3Bucket.

Drop the following components onto the design area:

tprejob	tLoop x 3	tFileDelete x 2
tFileInputDelimited x 2	tJava	tS3Connection
tContextLoad	tDBInput	tS3Put
tDBConnection	tFileOutputDelimited	
tSetGlobalVar	tFileOutputJSON	

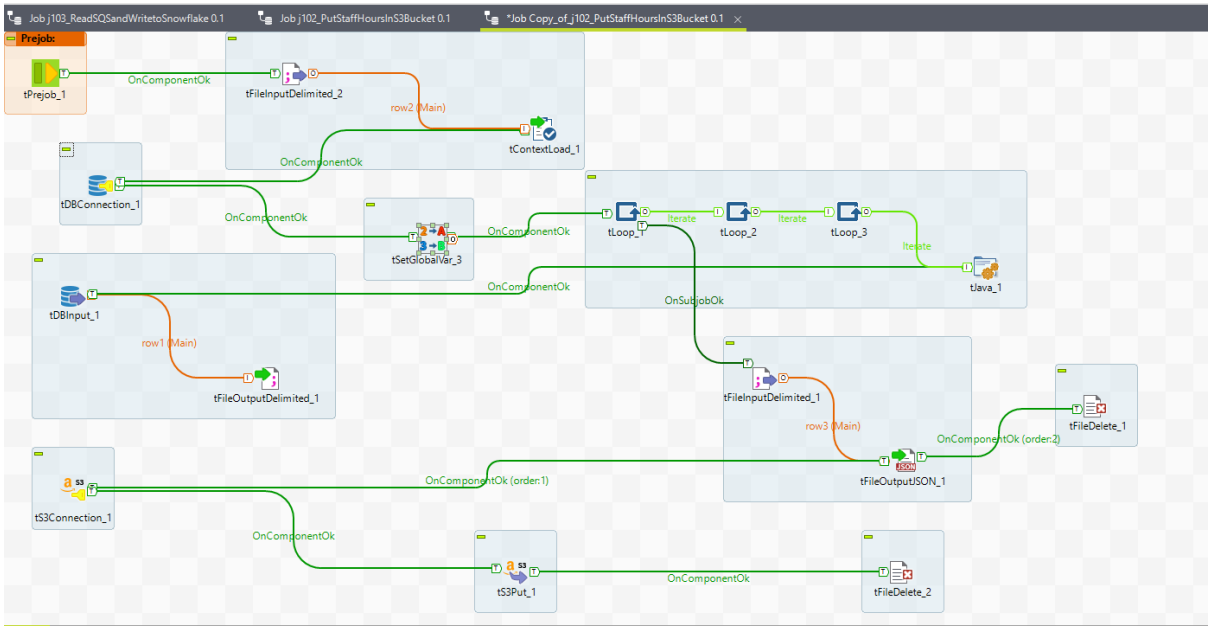
Arrange them in a similar layout to below.



Create the following connections between the components:

From Component	To Component	Join Type
tprejob	1st tFileInputDelimited	OnComponentOk trigger
1st tFileInputDelimited	tContextLoad	Row Main
tContextLoad	tDBConnection	OnComponentOk trigger
tDBConnection	tSetGlobalVar	OnComponentOk trigger
tSetGlobalVar	1st tLoop	OnComponentOk trigger
1st tLoop	2nd tLoop	Iterate
2nd tLoop	3rd tLoop	Iterate
3rd tLoop	tJava	Iterate
tJava	tDBInput	OnComponentOk trigger
tDBInput	tFileOutputDelimited	Row Main
1st tLoop	2nd tFileInputDelimited	OnSubjobOk trigger
2nd tFileInputDelimited	tFileOutputJSON	Row Main
tFileOutputJSON	1st tFileDelete	OnComponentOk trigger
tFileOutputJSON	tS3Connection	OnComponentOk trigger
tS3Connection	tS3Put	OnComponentOk trigger
tS3Put	2nd tFileDelete	OnComponentOk trigger

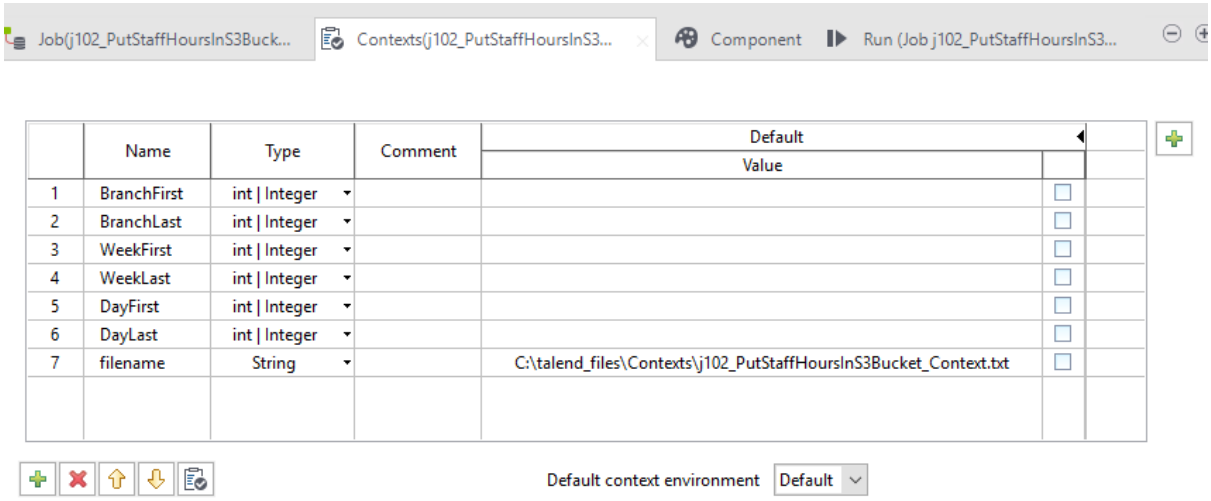
The resulting diagram should be similar to the following example:



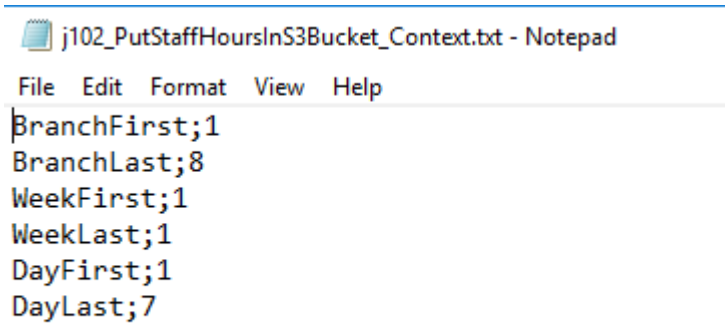
Configure the components as follows:

tPrejob requires no configuration and just initiates a task each time the job starts and is guaranteed to run prior to the main job.

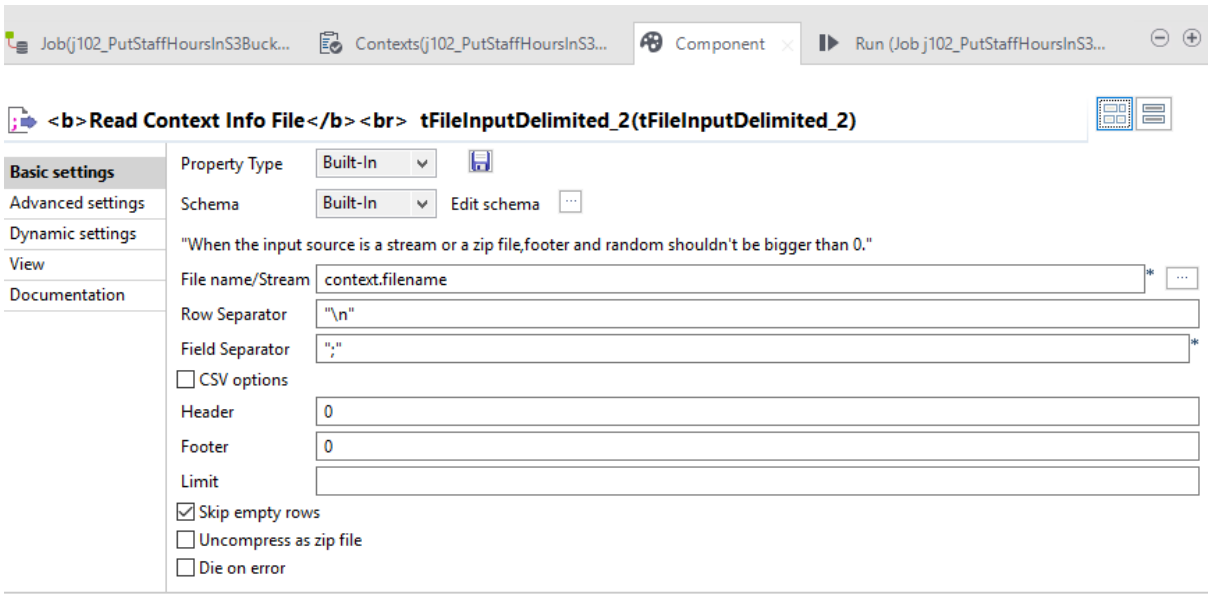
1st **tFileInputDelimited** reads in data used to populate the context variables



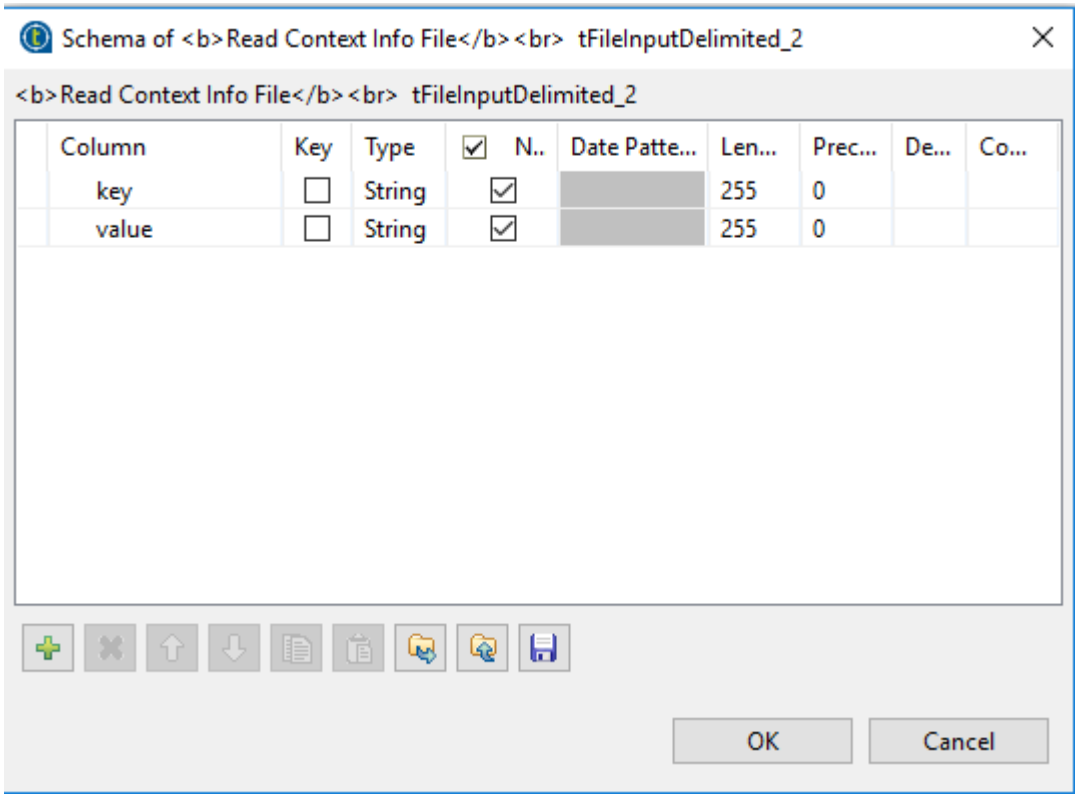
Using a file in the following format:



Select the component in the design area and click the **Component** tab to display the editor.

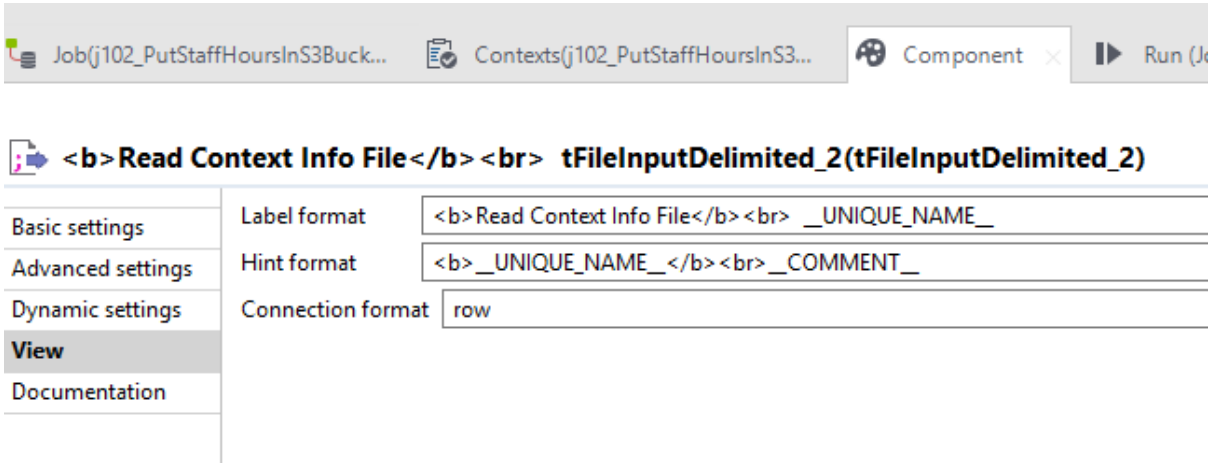


Use the context value of **filename** to identify the file and use the default values “\n” and “;” for row and field separators. There are zero header and footer lines and skip empty rows. Click on the **Edit Schema** ellipsis to check the definition.



The schema should consist of a key value pair both of string type. Press **OK** to close the editor.

Select the **View** option in the menu to display the label format.

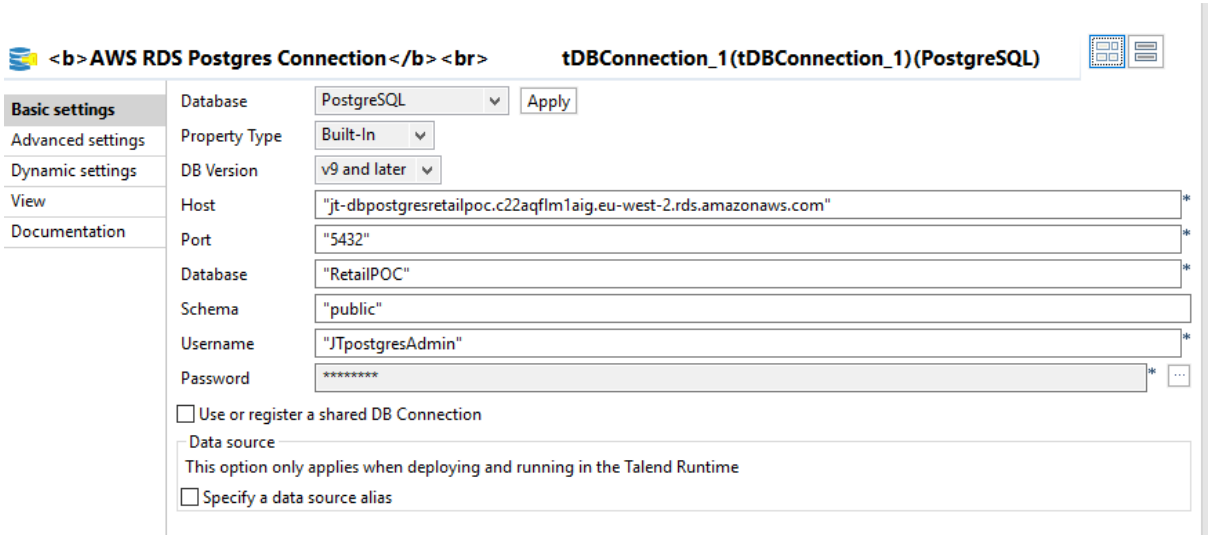


Modify the label format from __UNIQUE_NAME__ to **Read Context Into File**
 __UNIQUE_NAME__ . This will give a user-friendly label to the component but also retain the component name to assist with monitoring. Labelling all subsequent components should follow a similar pattern.

tContextLoad takes the file data imported in the previous component and uses it to populate the context variables named in the file. No configuration is necessary on this component as the default values are sufficient.

Label the component "**Load File Data To Context**" using the method previously shown.

tDBConnection is used to connect to the Postgresql instance which is running in AWS RDS as configured previously.



Select **PostgreSQL** as the database with version v9 or later. The Host value will be the URL shown in the AWS RDS console for the instance. Port should be the standard value 5432 unless a different setup has been used. The Database name will be **RetailPOC** and the public schema is used. Finally, Username and Password refer to the local user set up in RDS for the instance. A shared connection is not used in this case and since the job will just be ran in Studio a data source alias is not required either and both check boxes can remain unchecked.

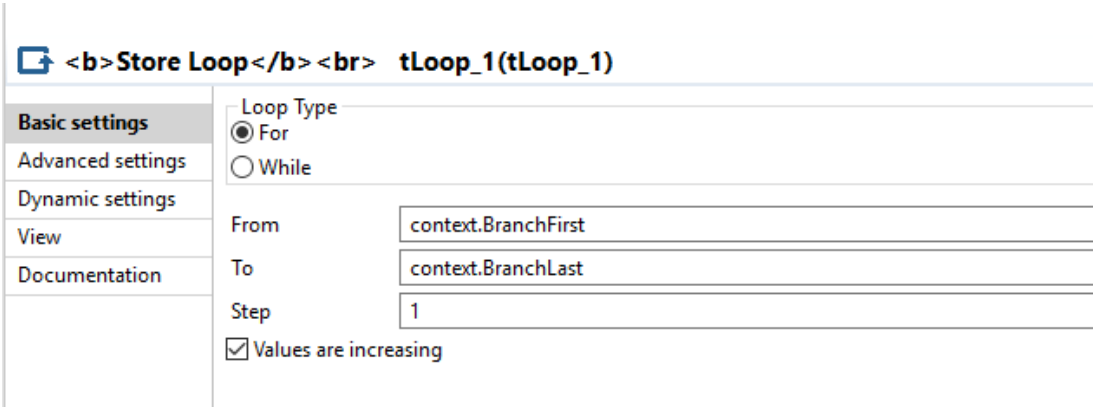
Go into the **view** option and label the component "**AWS RDS Postgres Connection**".

tSetGlobalVar initialises the variable for holding the current week number, day number and store code in the global cache. Click the component in the design area and select the **Component** tab in the bottom pane to show the editor.

From the Basic Settings option, press the **green + icon** to create a new row and name it "**vWeekNo**". Assign a default value of "". Repeat the process for **vDayNo** and **vStoreCode** and add a user-friendly label "**Initialise Globals**" to the component.

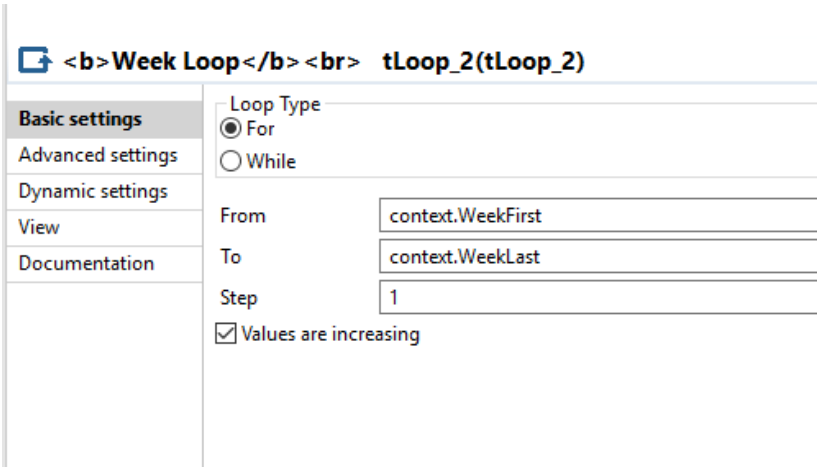
Three nested loops are used to generate the store staff data.

The first **tLoop** component iterates through the defined stores.



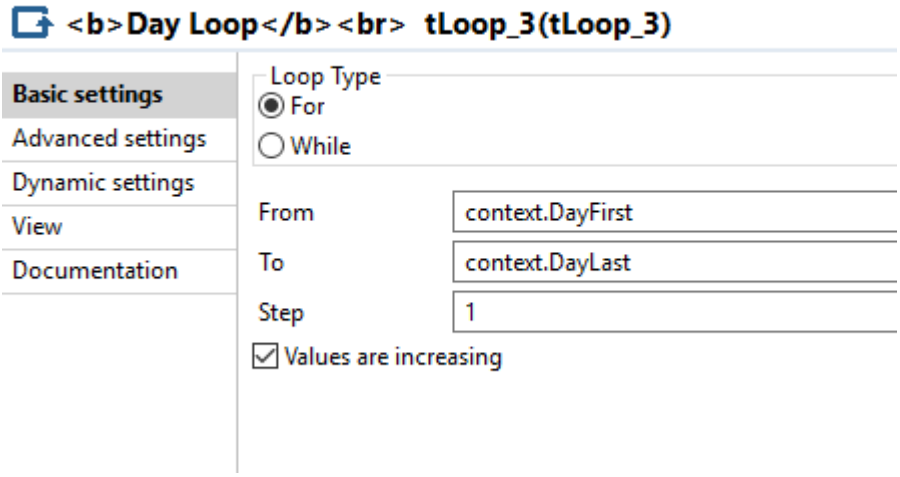
Select **For** loop as the type use the branch context values to define **From** and **To** parameters with a step of 1. Once complete modify the label to be "**Store Loop**"

The second **tLoop** processes the week values.



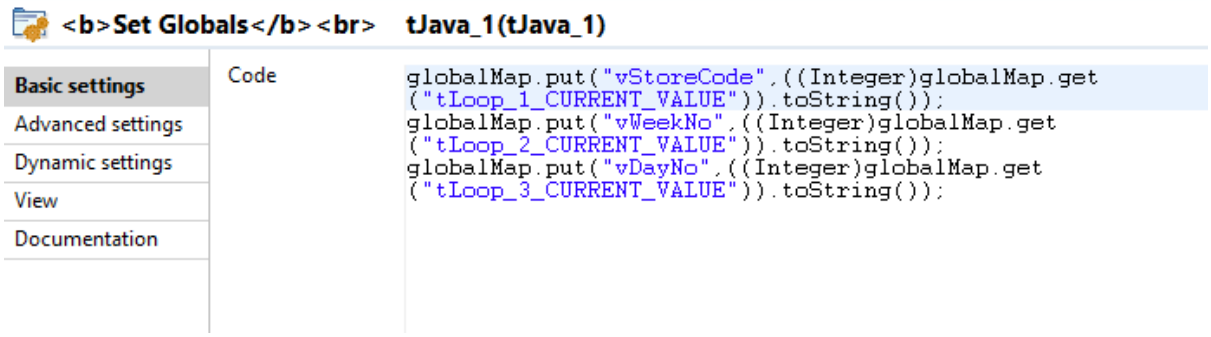
Similar in definition to the previous component but use the week start and end values from the context. Label the component "**Week Loop**".

The third **tLoop** traverses the day values.



Similar configuration again but select day values from the context. Label the component "**DayLoop**".

tJava is used to set the current store, week and day values into the global variables for each iteration of the inner loop.

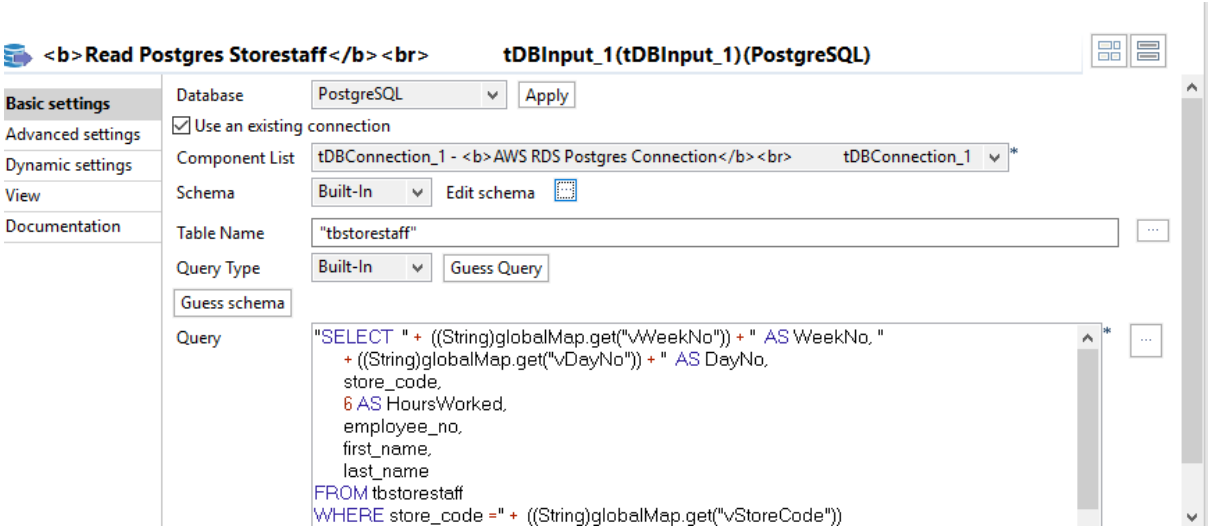


Assign **vStoreCode** the current value of the outer loop, **vWeekNo** the current value of the middle loop and **vDayNo** the value of the inner loop as shown in the diagram.

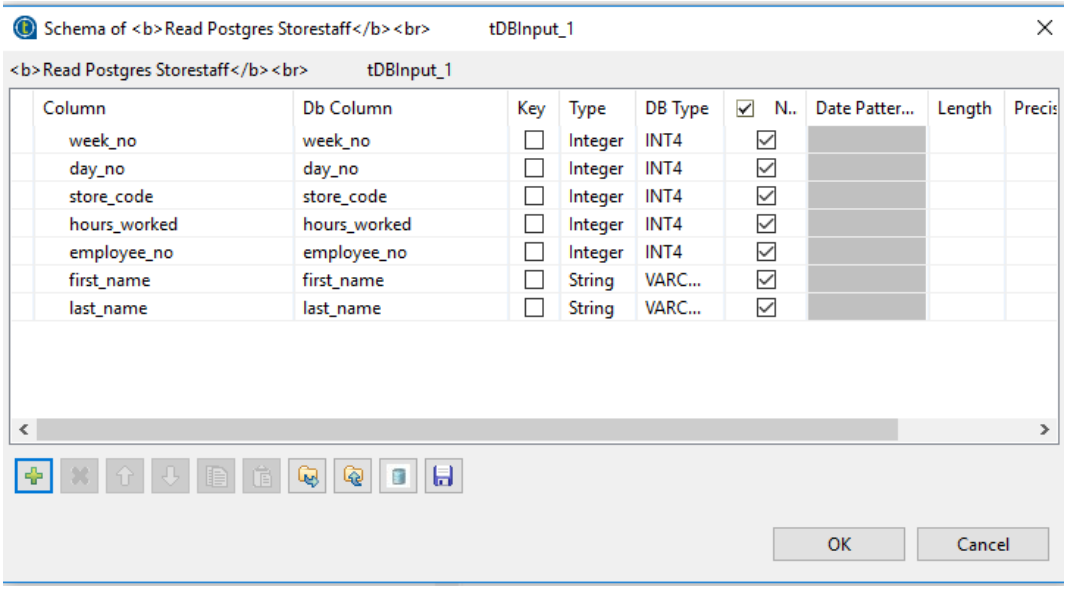
Make sure the CURRENT_VALUE property is used and not CURRENT_ITERATION. Although they will be the same if the loops all start from 1 this doesn't have to be the case. For example, if branches 3 to 4 only were specified in the context range then the for the first iteration the value of CURRENT_VALUE would be 3 but CURRENT_ITERATION has a value of 1.

Label the component "**Set Globals**".

tDBInput retrieves data from the PostgreSQL database by running a query incorporating the global variables to generate simulated sales hours for each store on a daily basis.



Select **PostgreSQL** as the database type and check the **Use an existing connection** option. Select the DB connection previously configured and the Table Name "**tbstorestaff**". Select **Edit schema** to call the editor.



Click the green plus icon to add each row and add the following entries to the schema.

Column	Type
week_no	Integer
day_no	Integer
store_code	Integer
hours_worked	Integer
employee_no	Integer
first_name	String
last_name	String

Press **OK** to close the schema editor the enter the following text into the Query field.

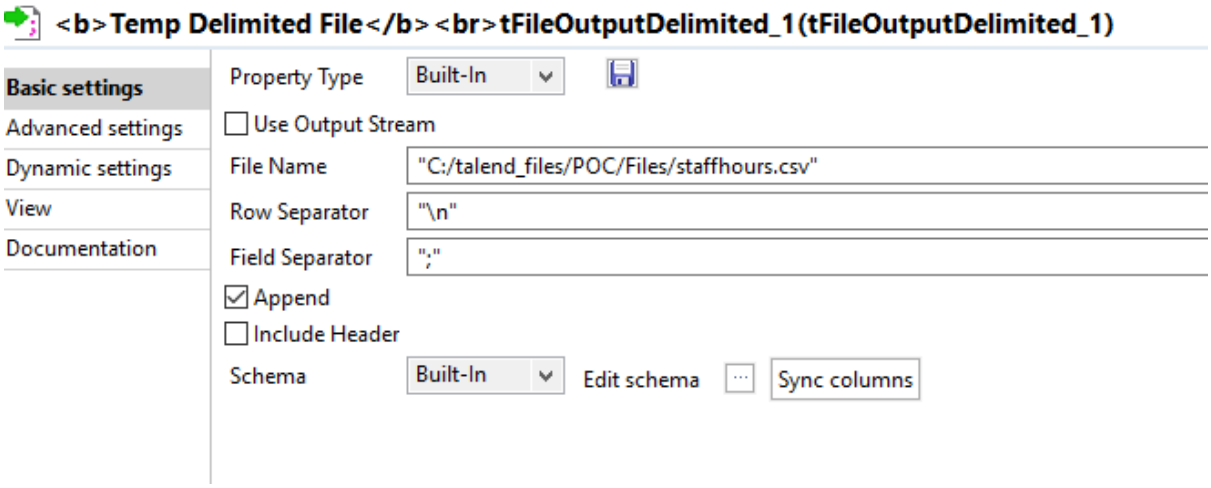
```
"SELECT  " +  ((String)globalMap.get("vWeekNo")) + "  AS WeekNo, "
       + ((String)globalMap.get("vDayNo")) + "  AS DayNo,
       store_code,
       6 AS HoursWorked,
       employee_no,
       first_name,
       last_name
FROM tbstorestaff
WHERE store_code =" +  ((String)globalMap.get("vStoreCode"))
+ " ORDER BY RANDOM()
LIMIT 10"
```

The query models the rules of the proof of concept. Each store is open 12 hours a day and has a full time equivalent of 5, representing a total of 60 hours. Staff work in 6 hour shifts therefore a total of 10 staff from a pool of 20 is required for each store per day.

Note the query is constructed using dynamic SQL to incorporate the global variables. There may be concerns about security vulnerability from SQL injection and if this was public facing such as a web site query I would agree. In this case it is only internal and doesn't present a risk but if concern remained it could be redesigned to use a parametrised stored procedure instead.

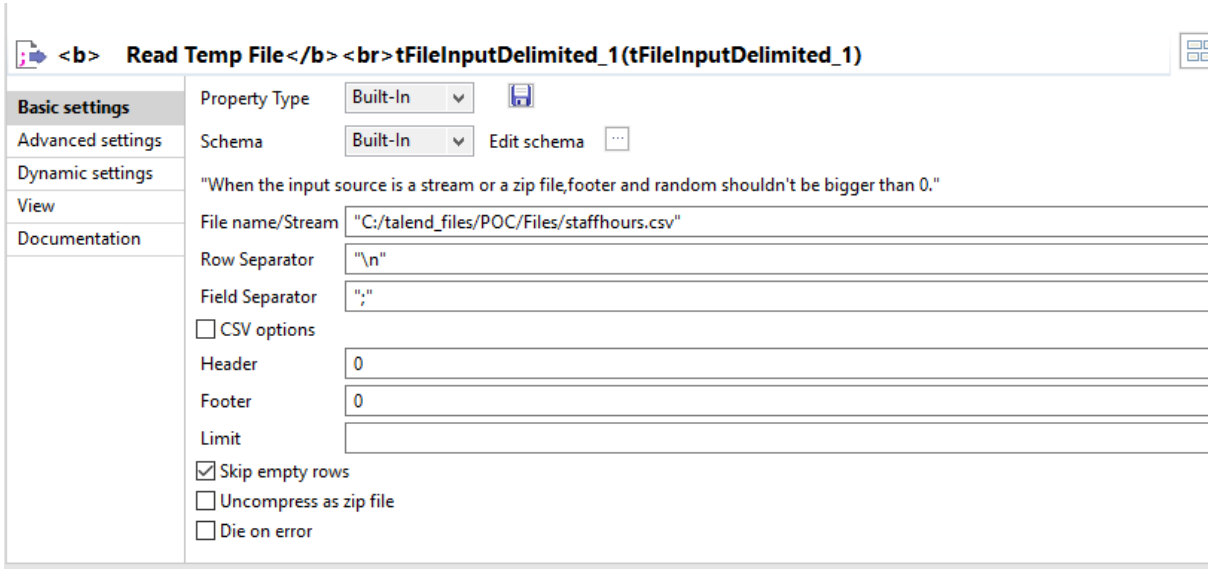
Label the component "**Read Postgres Storestaff**".

tFileOutputDelimited creates a temporary output file used to store the data retrieved from PostgreSQL. Ultimately the data will be written to a JSON format file however it is created using an iterative method requiring thew output file to be regularly appended. JSON files in Talend don't have this option and would be overwritten with each iteration therefore the temporary delimited file which can be appended is used instead and this file used later to create the JSON document in a single pass.



Enter the filename and leave the separators with the default values. Click on **Edit schema** to ensure it is the same as the tDBInput component. Ensure that the Append option is checked then label the component "**Temp Delimited File**".

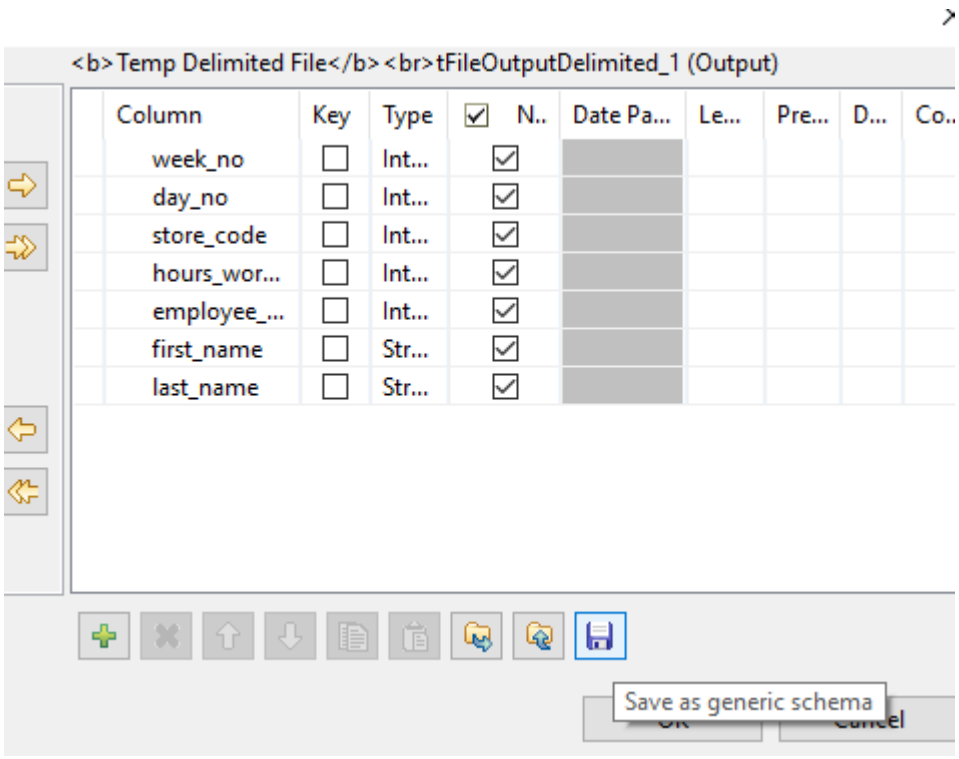
The second **tFileInputDelimited** component is used to read the completed temporary file from the previous section. Initiation is by an **OnSubjobOk** trigger from the outer **tLoop** component. Using this type of trigger ensures that the loop iterations are fully complete including the writing of the temporary file, prior to it being consumed.



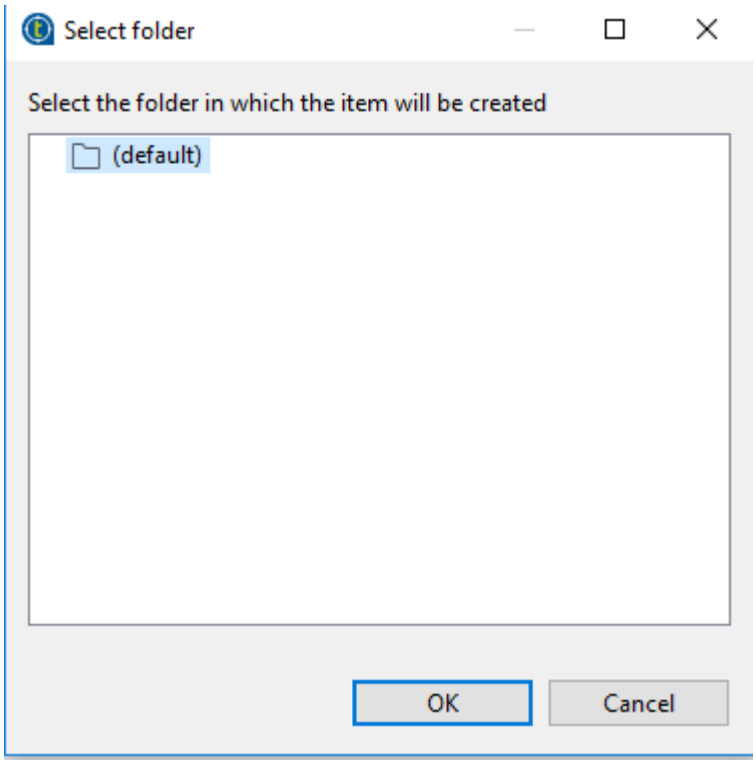
The schema will need to match the temporary file, and this can be achieved by clicking the ellipsis and manually entering the values. An alternative way that may be easier and avoid cumulative errors

is to save the schema from the **tFileOutputDelimited** component to the **repository** as a **generic schema**, which can then be used to set the schema on this component.

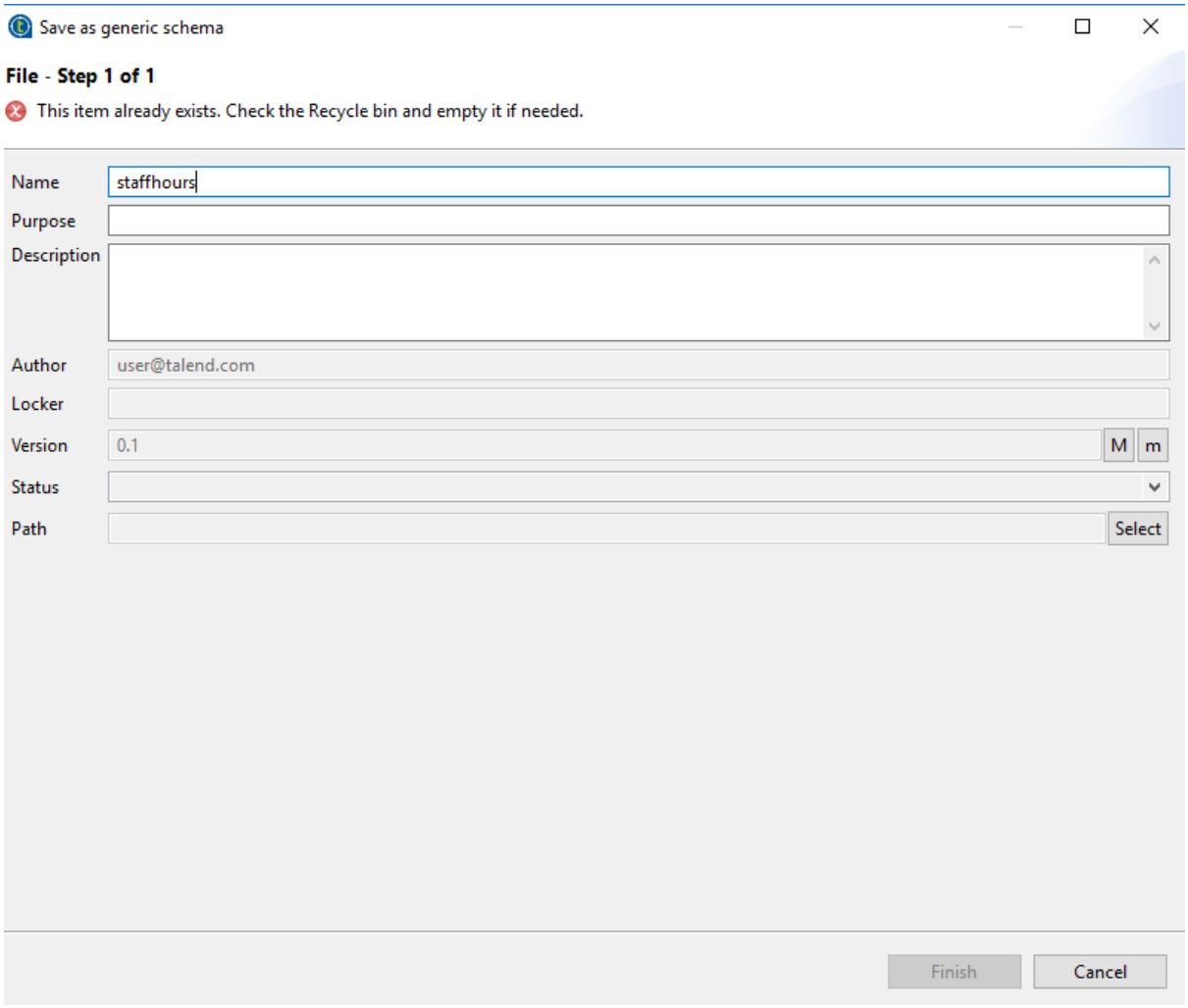
To do this return to the **tFileOutputDelimited** component and click the ellipsis to enter the schema editor.



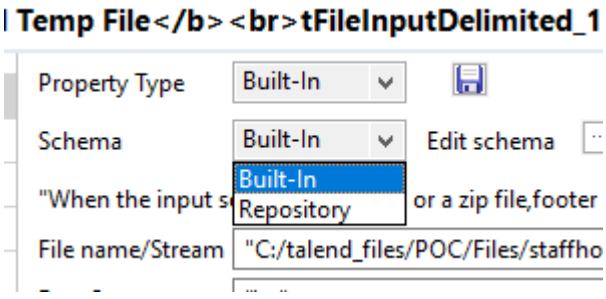
Press the disk icon to Save as generic schema



Click **OK** to accept the default location which equates to Generic schemas folder in the Metadata section of the repository.

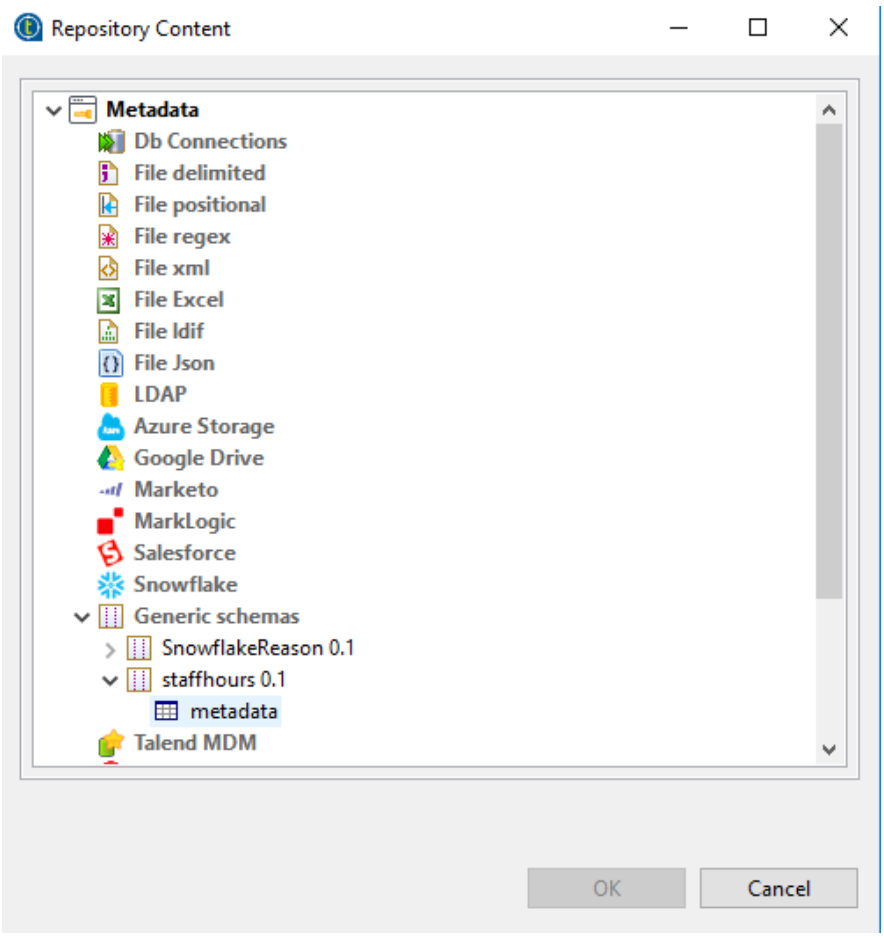


Name the schema "**staffhours**" and click **OK** to complete, then return to the second **tFileInputDelimited** component.

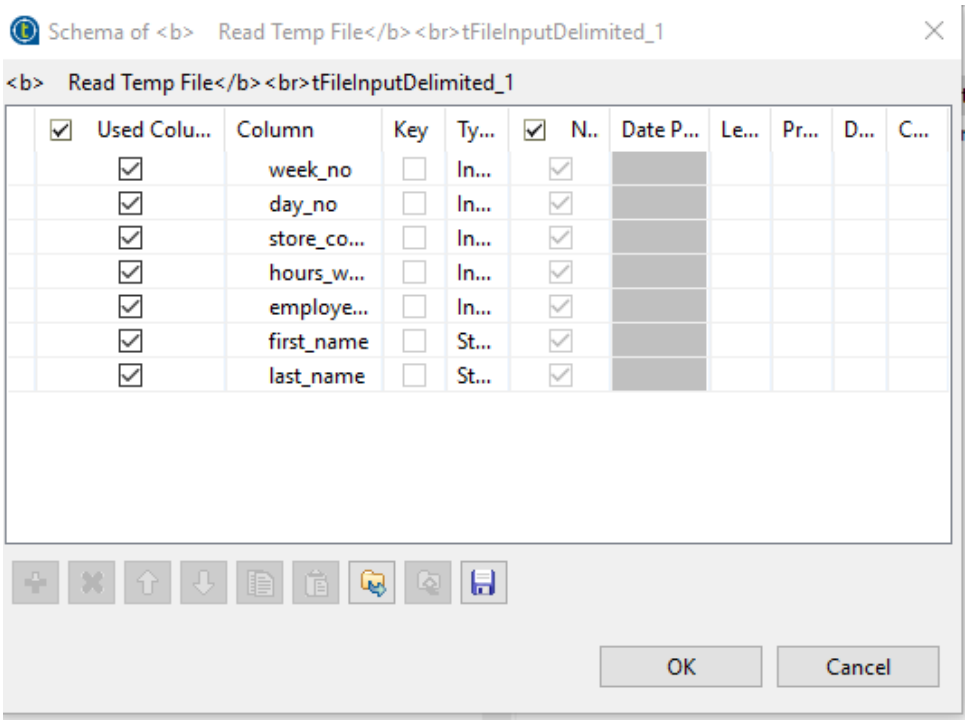


Select the **Repository** option from the Schema drop-down list which will add an edit control for the repository schema name.

Click the ellipsis next to the control to call the repository viewer.

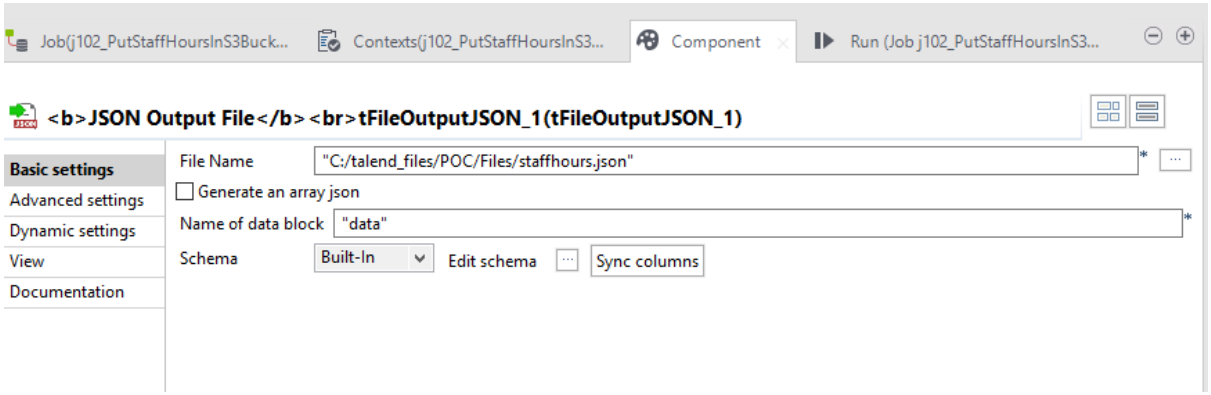


Expand the Generic schemas section. Select staffhours and click **OK** to complete. You can check the schema by clicking **Edit schema** and selecting the **View Schema** option.



File name should match the **tFileOutputDelimited** component and the separators will be the default values. There are no header or footer rows and empty rows can be skipped. Label the component "**Read Temp File**".

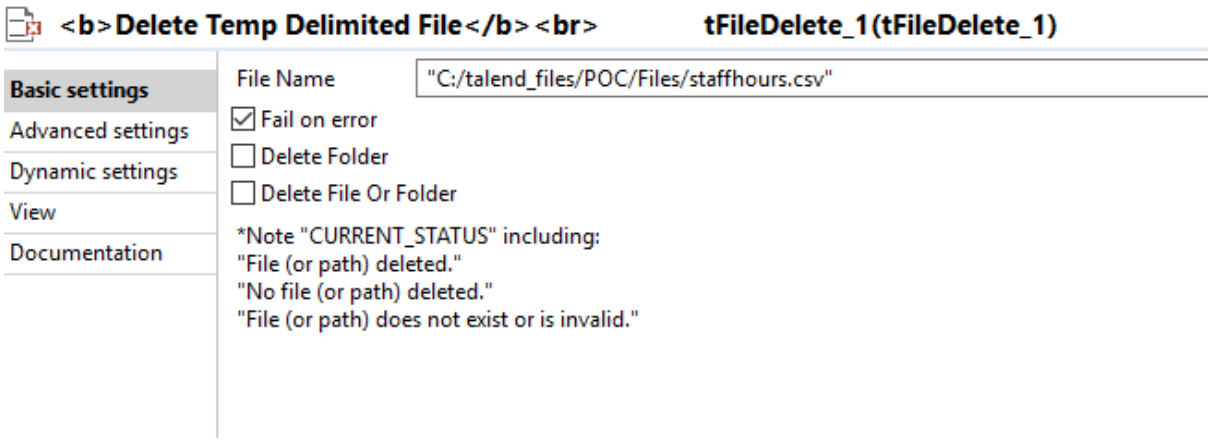
tFileOutputJSON will output the data read from the temporary file in JSON format. A very basic format is used for this POC with the fields of each record presented in a single data block with no nesting.



Inset the filename which will be an initial location prior to being stored in an S3 bucket. The name of the data block can be left as "**data**" and ensure the schema matches the input file by pressing **Sync columns**.

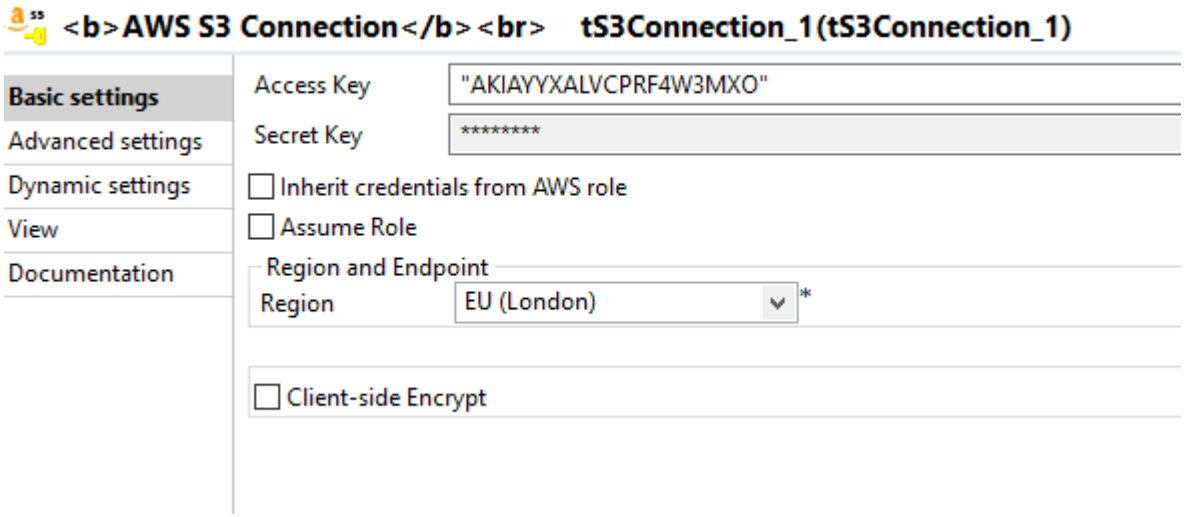
Label the component "**JSON Output File**".

The first **tFileDelete** component removes the temporary delimited file after the JSON file has been created.



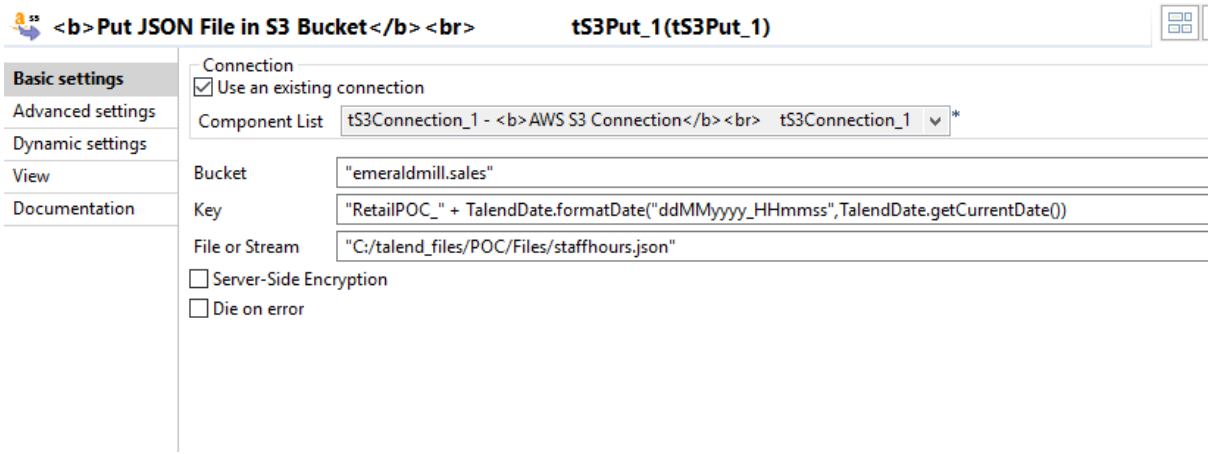
Enter the file name and label the component "**Delete Temp Delimited File**".

Connection to the S3 bucket is initiated by the **tS3Connection** component.



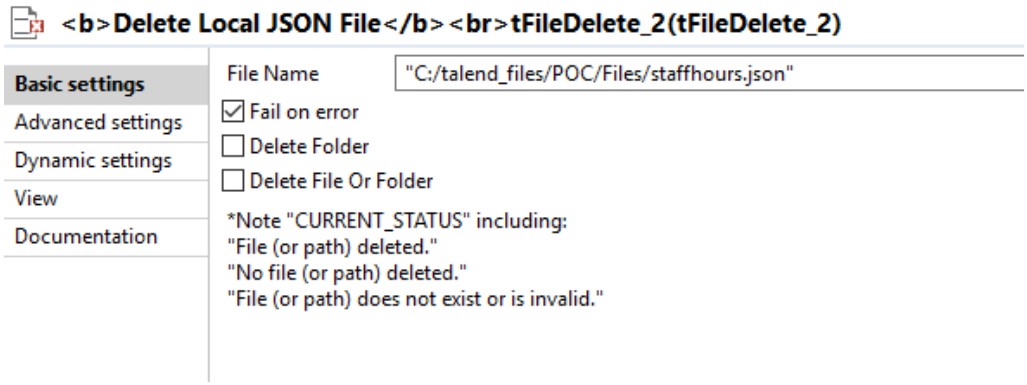
Use the keys generated when creating the bucket, as described earlier in this document and select the AWS Region which should match the bucket location. Label the component "**AWS S3 Connection**"

Having defined the connection**, tS3Put** is used to move the JSON file to the bucket.



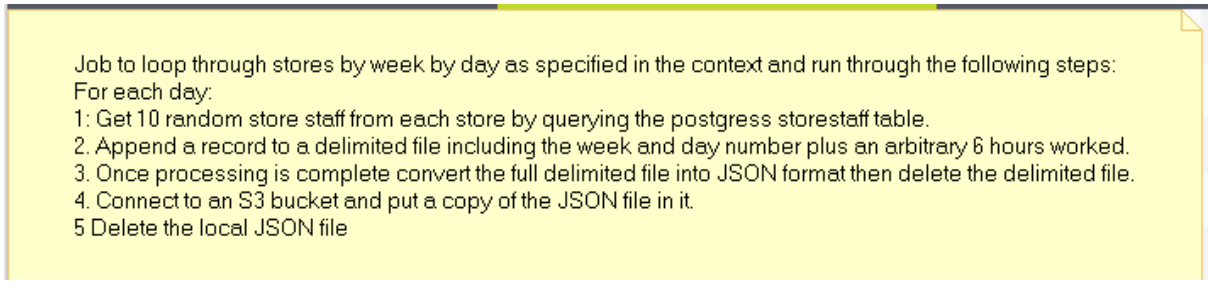
Check the Use an existing connection option and select the **S3 connection** from the list. Add the bucket name and the Key which is the S3 equivalent of file name. To prevent over writing a timestamp will be concatenated onto the name using Talend date functions. The key will take the form "RetailPOC_" + TalendDate.formatDate("ddMMyyyy_HHmmss",TalendDate.getCurrentDate()). Enter the location of the JSON file and label the component "**Put JSON File in S3 Bucket**".

The final action of the job is to remove the local JSON file after a copy has been stored in S3, using a second **tFileDelete** component.

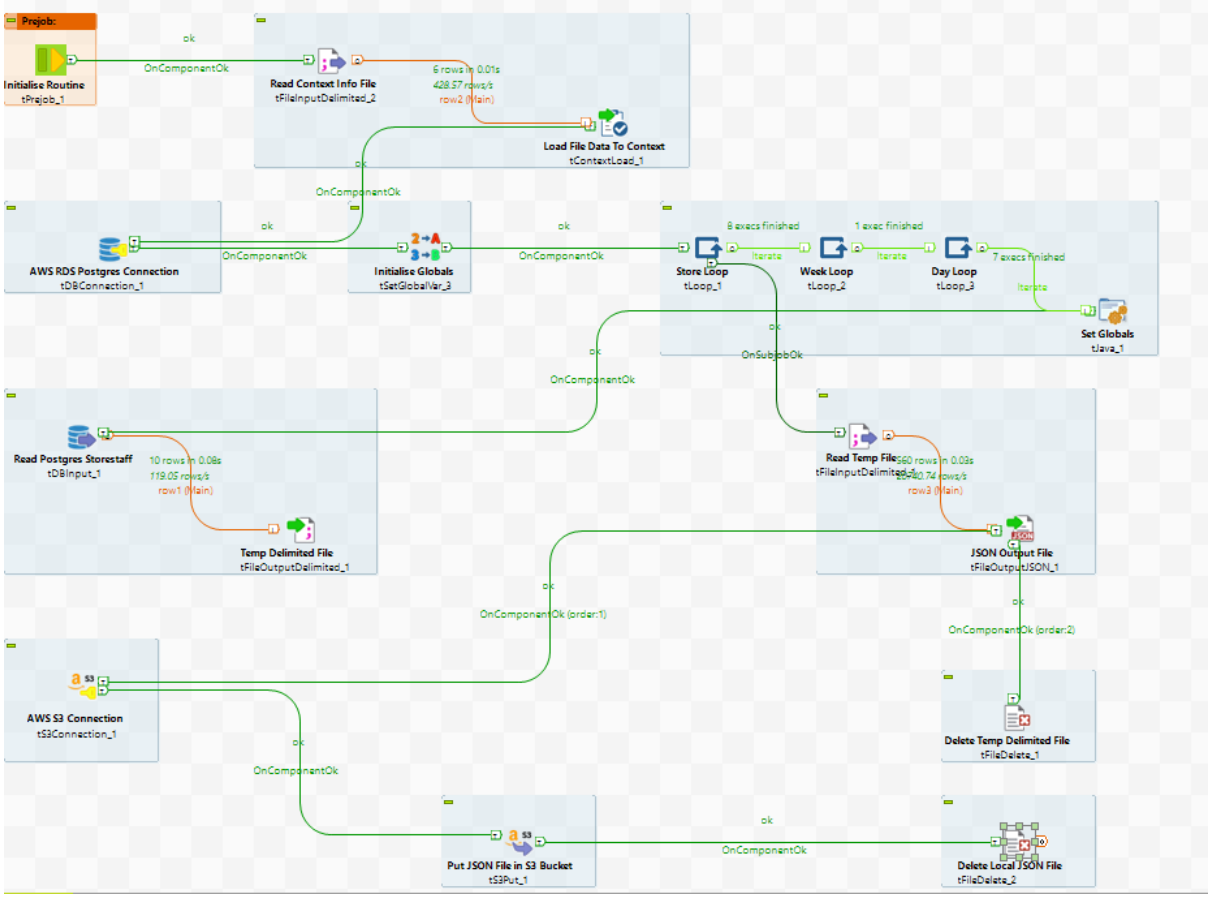


Enter the location of the JSON file and label the component "**Delete Local JSON File**"

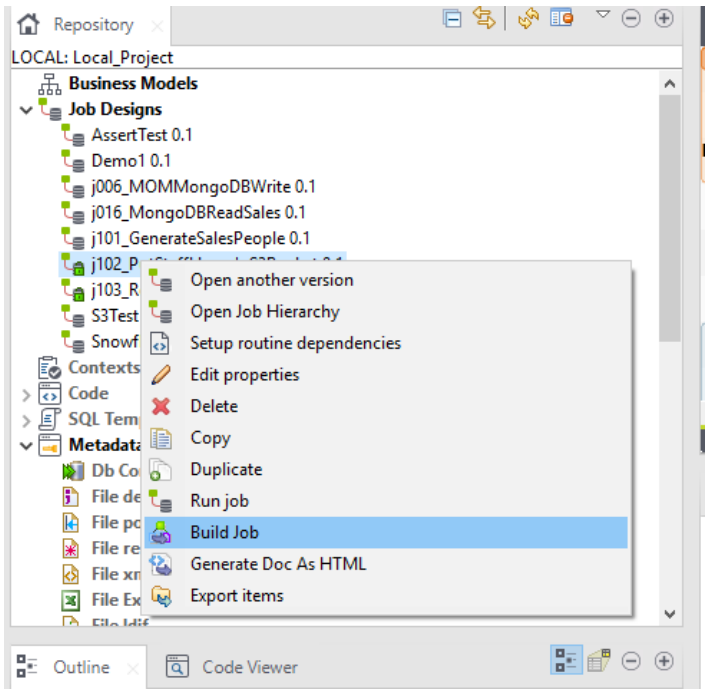
To complete the job, add a note to the design area similar to the following:



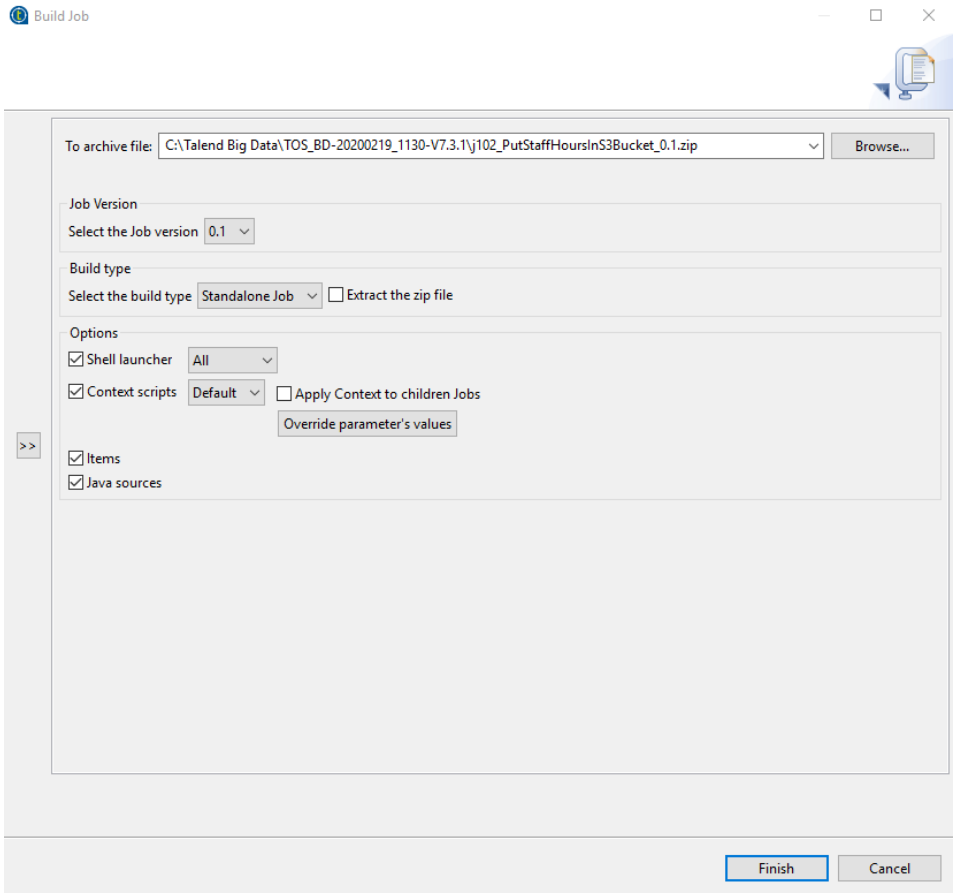
The finished procedure will look similar to the illustration below.



Unlike the previous job, which was just ran in Open Studio, this task will be deployed onto the AWS EC2 instance and ran as a stand-alone job. To begin the build, save the job then **right click** on its name under **Job Designs** in the repository pane and select **Build Job**.



The Build Job options will be displayed and in this case all default values are acceptable.



Click on **OK** to begin the build and once complete a zip file will be created in the location specified. In this case that is the root folder of Talend Studio although that can be changed if required.

13. Create job to write to Snowflake

The third Talend job responds to messages arriving on the SQS queue. Messages are created by the AWS Lambda function that responds to files being placed in the S3 bucket by the previous job. After initiation this process will retrieve the file key from the message body and use it to download the JSON file from the S3 bucket. On receiving the file, the data, simulated staff timesheets, will be loaded into Snowflake for analysis later. The local JSON file will then be archived.

Create a new job in the repository called j103_ReadSQSandWritetoSnowflake.

Drop the following components onto the design area:

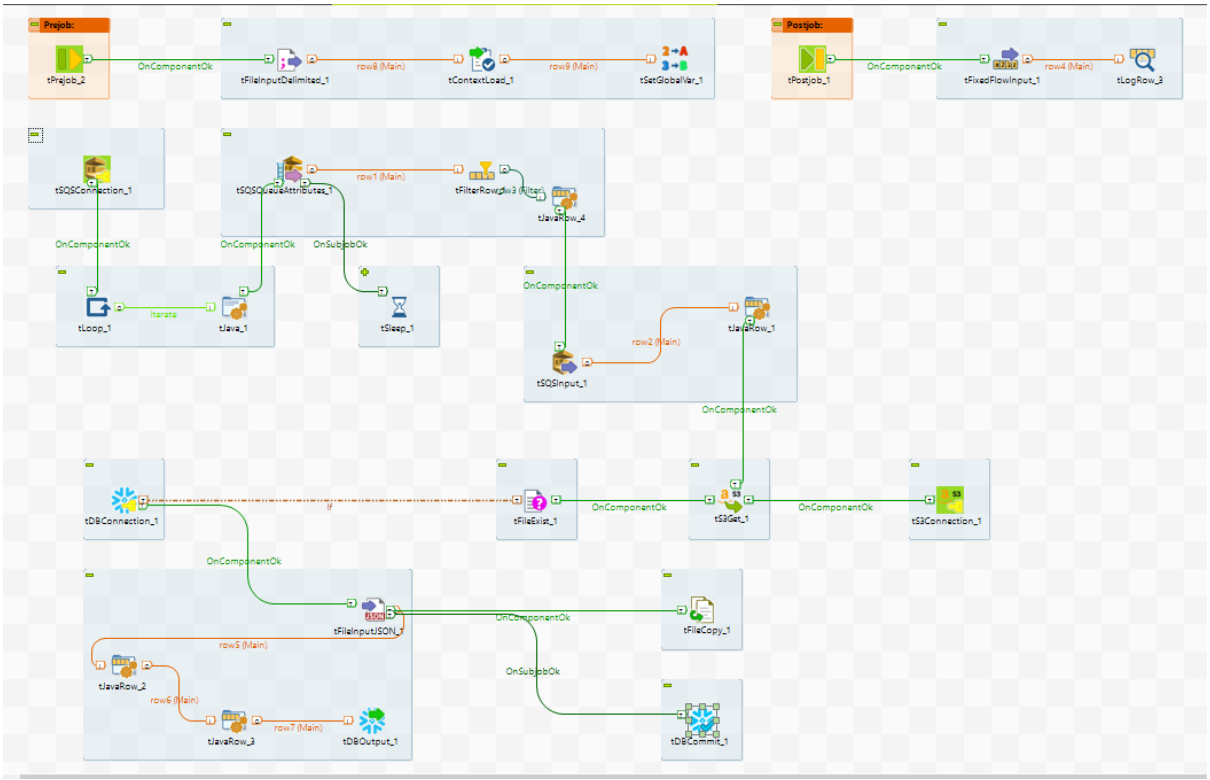
tPrejob	tLoop	tS3Get
tFileInputDelimited	tJava	tFileExist
tContextLoad	tSQSQueueAttributes	tDBConnection
tSetGlobalVar	tSleep	tFileInputJSON
tPostjob	tFilterRow	tDBOutput
tFixedFlowInput	tJavaRow X 4	tFileCopy
tLogRow	tSQSInput	tDBCommit
tSQSConnection	tS3Connection	

Arrange the components in a similar way to the following diagram.

Now create the joins between the components according to the following table.

From Component	To Component	Join Type
tPrejob	tFileInputDelimited	OnComponentOk trigger
tFileInputDelimited	tContextLoad	Row Main
tContextLoad	tSetGlobalVar	Row Main
tPostjob	tFixedFlowInput	OnComponentOk trigger
tFixedFlowInput	tLogRow	Row Main
tSQSConnection	tLoop	OnComponentOk trigger
tLoop	tJava	Iteration
tJava	tSQSQueueAttributes	OnComponentOk trigger
tSQSQueueAttributes	tSleep	OnSubjobOk trigger
tSQSQueueAttributes	tFilterRow	Row Main
tFilterRow	1st tJavaRow	Row Filter
1st tJavaRow	tSQSInput	OnComponentOk trigger
tSQSInput	2nd tJavaRow	Row Main
2nd tJavaRow	tS3Get	OnComponentOk trigger
tS3Connection	tS3Get	OnComponentOk trigger
tS3Get	tFileExist	OnComponentOk trigger
tFileExist	tDBConnection	Run if trigger
tDBConnection	tFileInputJSON	OnComponentOk trigger
tFileInputJSON	tFileCopy	OnComponentOk trigger
tFileInputJSON	3rd tJavaRow	Row Main
3rd tJavaRow	4th tJavaRow	Row Main
4th tJavaRow	tDBOutput	Row Main
tFileInputJSON	tDBCommit	OnSubjobOk trigger

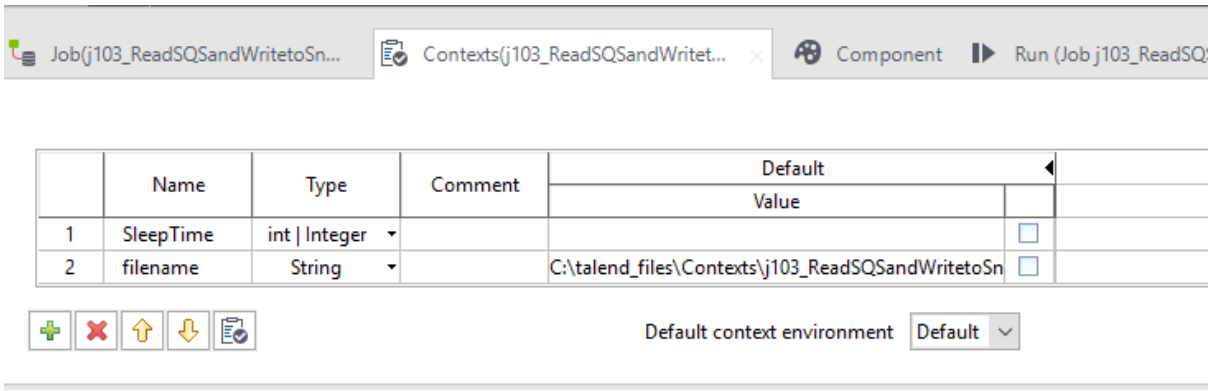
Once complete the job in the design area should resemble the following illustration.



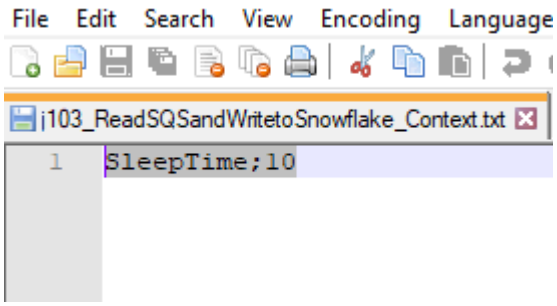
The components can now be configured using the following steps:

tPrejob requires no configuration and just initiates a task each time the job starts and is guaranteed to run prior to the main job.

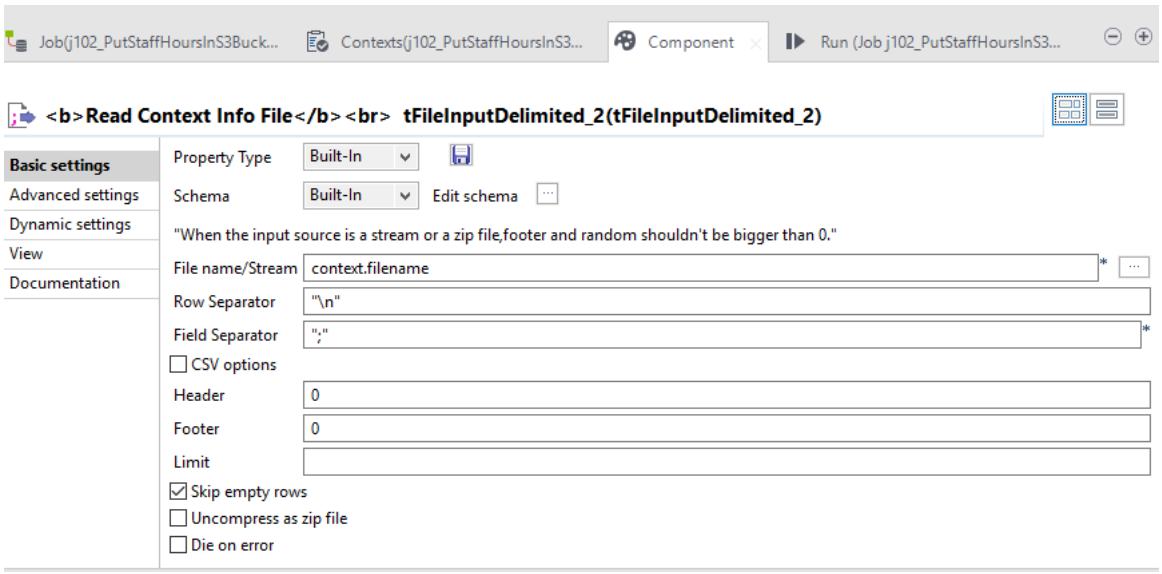
1st **tFileInputDelimited** reads in data used to populate the context variables



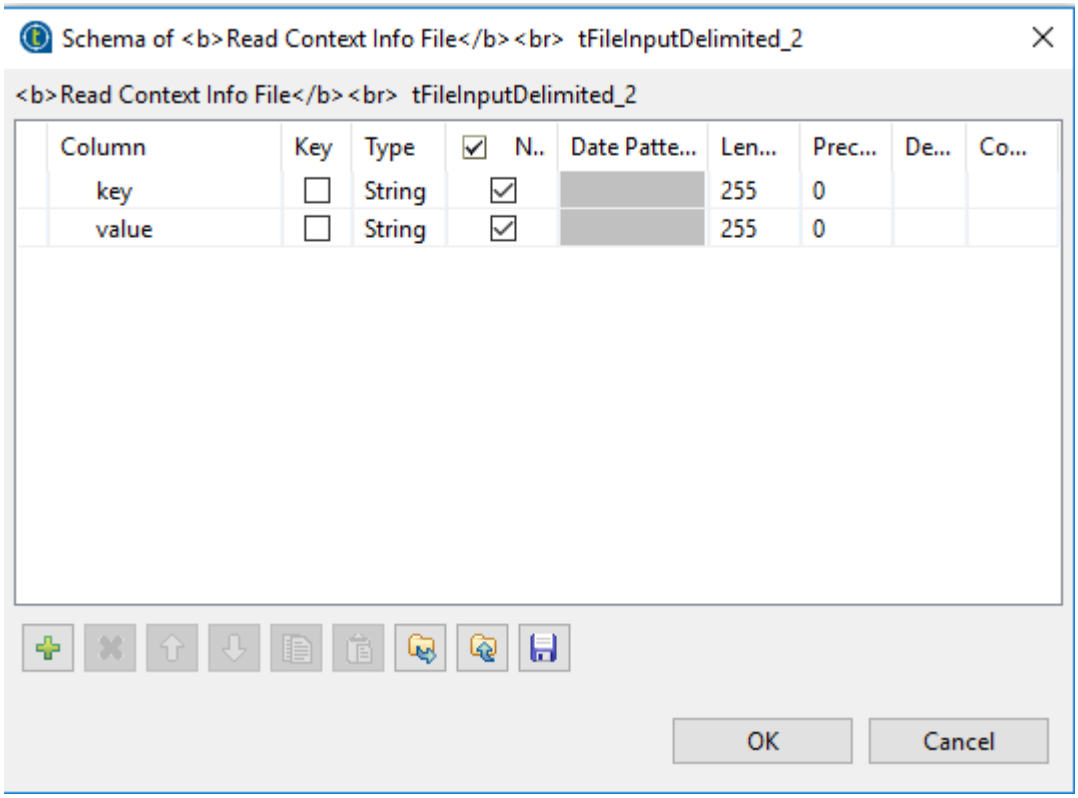
Using a file in the following format:



Select the component in the design area and click the **Component** tab to display the editor.

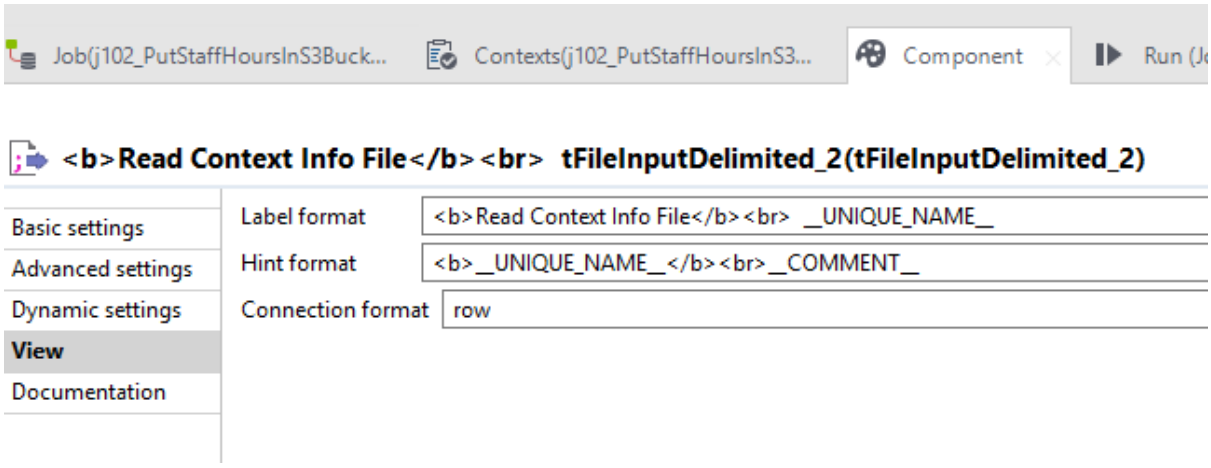


Use the context value of filename to identify the file and use the default values “\n” and “;” for row and field separators. There are zero header and footer lines and skip empty rows. Click on the **Edit Schema** ellipsis to check the definition.



The schema should consist of a key value pair both of string type. Press **OK** to close the editor.

Select the **View** option in the menu to display the label format.



Modify the label format from __UNIQUE_NAME__ to **Read Context Info File**
 __UNIQUE_NAME__ . This will give a user-friendly label to the component but also retain the component name to assist with monitoring. Labelling all subsequent components should follow a similar pattern.

tContextLoad takes the file data imported in the previous component and uses it to populate the context variables named in the file. No configuration is necessary on this component as the default values are sufficient.

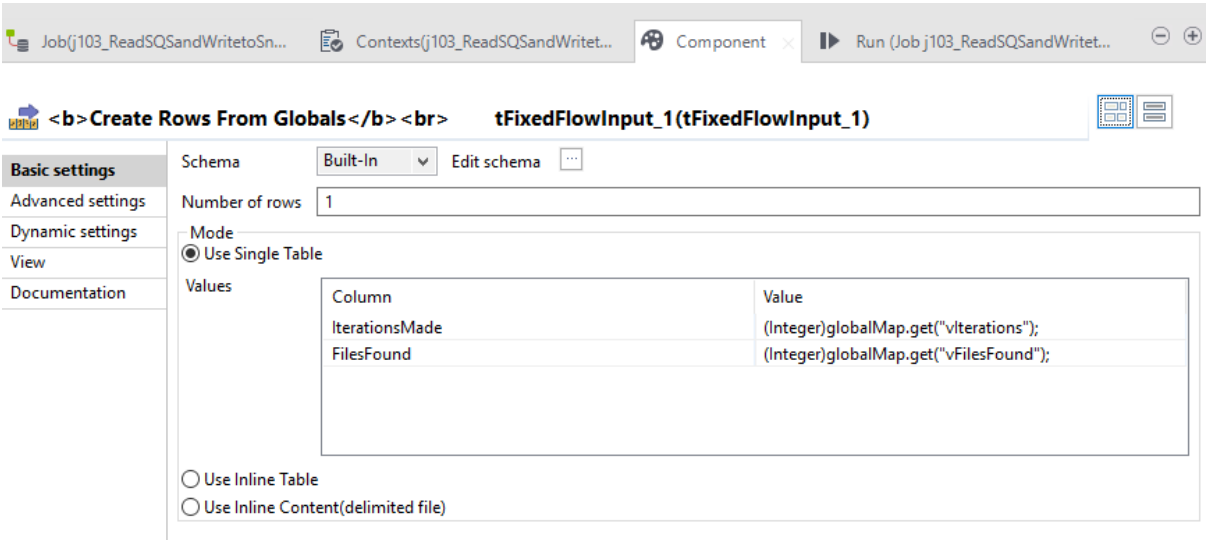
Label the component "**Load File Data To Context**" using the method previously shown.

tSetGlobalVar initialises the variable for holding the number of iterations, file found status and S3 bucket key in the global cache. Click the component in the design area and select the **Component** tab in the bottom pane to show the editor.

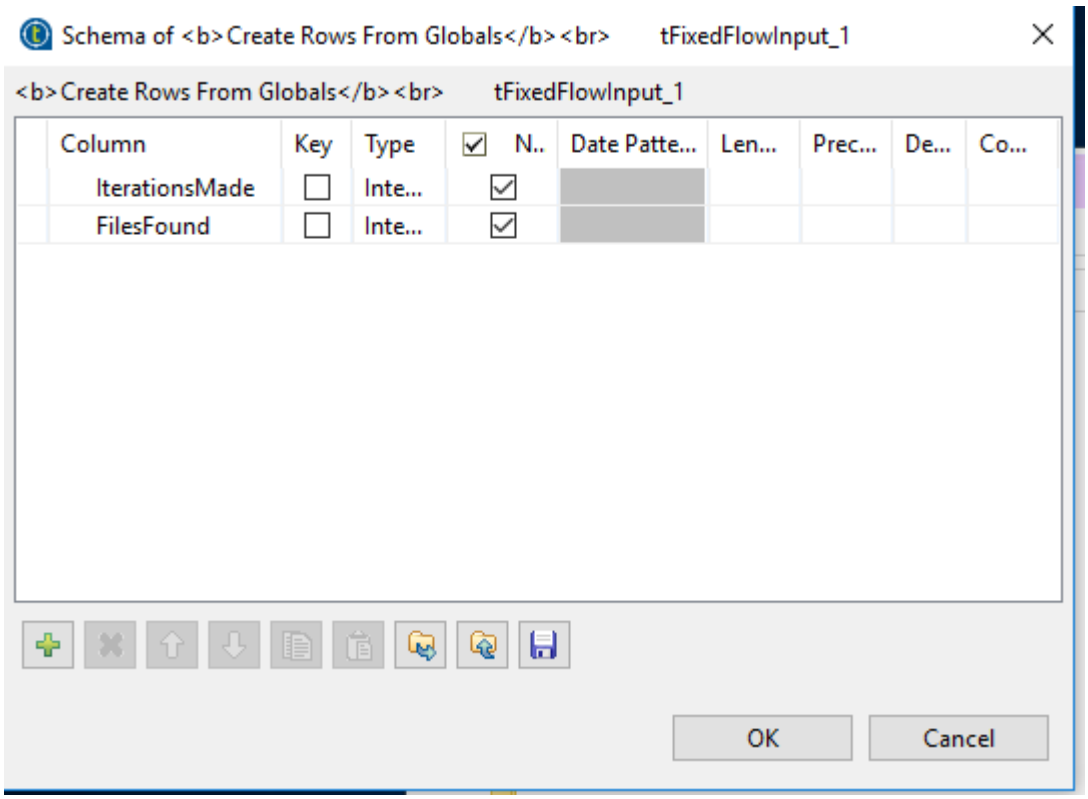
From the Basic Settings option, press the **green + icon** to create a row and name it "**vIterations**". Assign a default value of 0. Repeat the process for "**vFilesFound**" and "**vBucketKey**" which has a default value of "" and add a user-friendly label "**Initialise Globals**" to the component.

tPostjob requires no configuration initiating a task each time the job completes and is guaranteed to run after the main job. Since the job runs on an infinite loop this component and its associated sub job will only be called in the event of termination, but it is useful for debugging.

tFixedFlowInput generates a data flow using the values of the global variables.



Click the **Edit schema** ellipsis to create the schema.

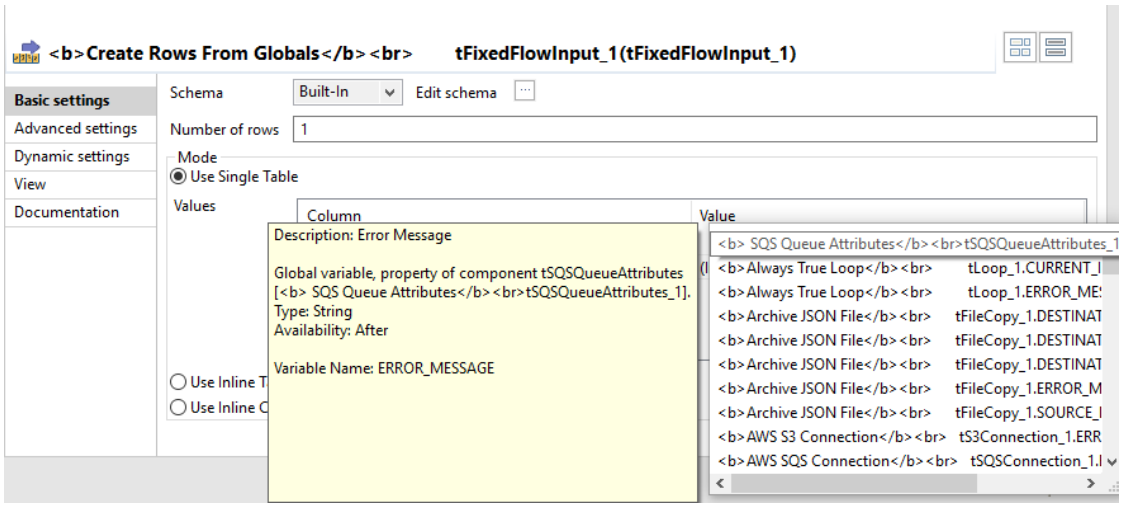


Click the **green + icon** to add 2 rows for "**IterationsMade**" and "**FilesFound**", both of integer type. Press **OK** to exit the editor.

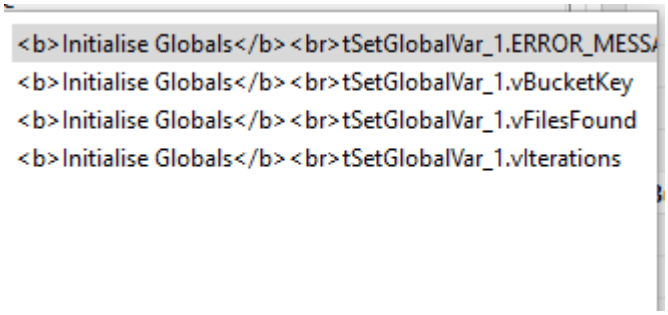
Select single table mode and assign the following values to the columns:

Column	Value
IterationsMade	((Integer)globalMap.get("vIterations"))
FilesFound	((Integer)globalMap.get("vFilesFound"))

A more reliable way to enter the global variable values is to allow Talend to look them up, avoiding transcription errors. Place the cursor in the value field and press CTRL and Enter simultaneously to bring up a list of system variables.

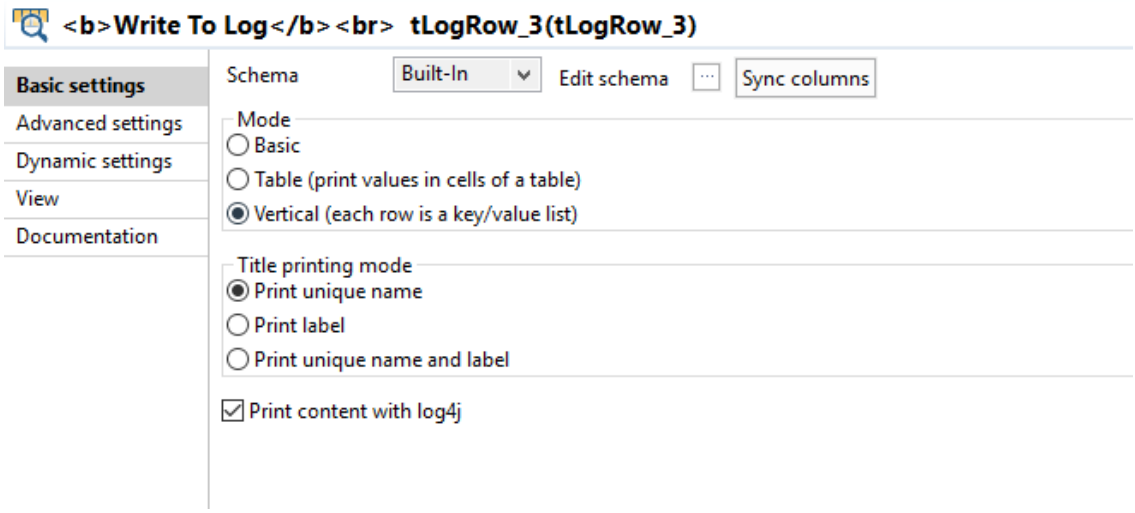


Scrolling through the list will eventually find the correct variable but a filter can be applied to make the process easier. Find the component name where the variable is initialised which in this case is **tSetGlobalVar**. When the list is displayed type "**tse**" (case insensitive) and the list will reduce to the entries applicable to that component only including the global variables.



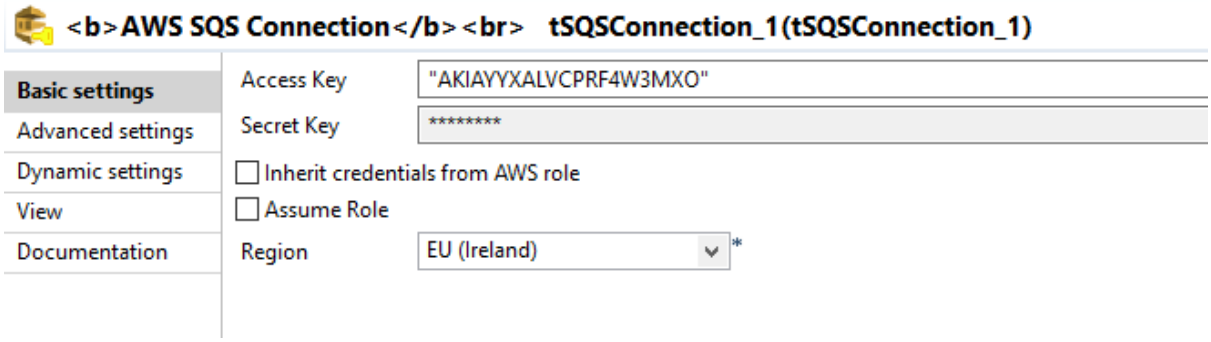
Move the cursor to the required entry and click to insert the value. Notice that by default it always casts the variable to a String. To change the value to an integer simply overtype the cast from String to Integer. Repeat the process for both global variables and label the component "**Create Rows From Globals**".

A **tLogRow** component is used to display the data flow values on the standard output.

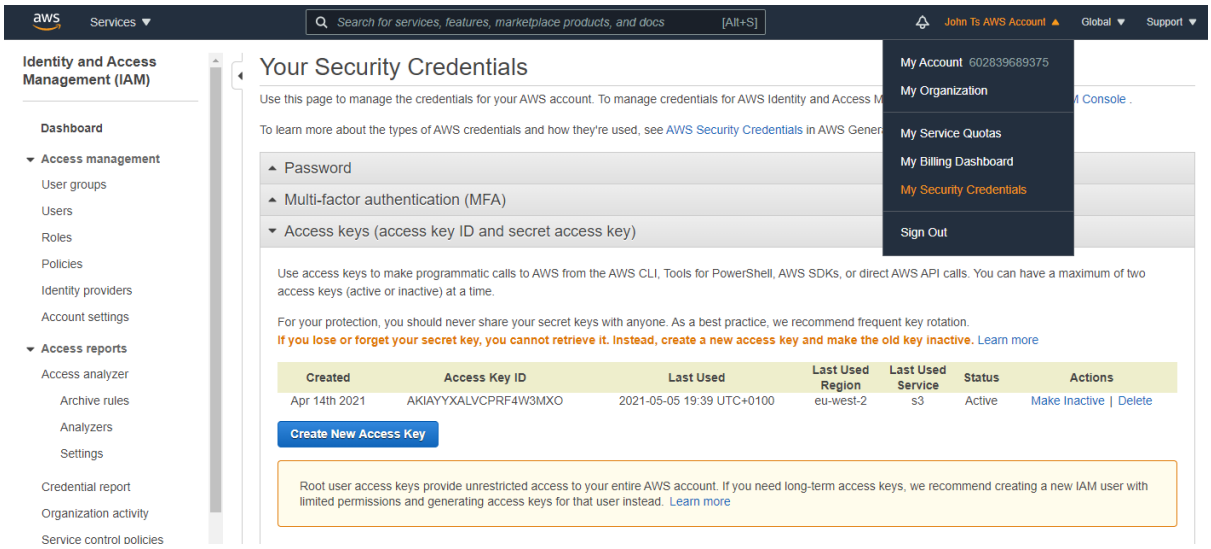


Select the **Vertical** mode option for clarity and label the component "**Write To Log**".

Connection to the AWS SQS queue is configured via the **tSQSConnection** component.

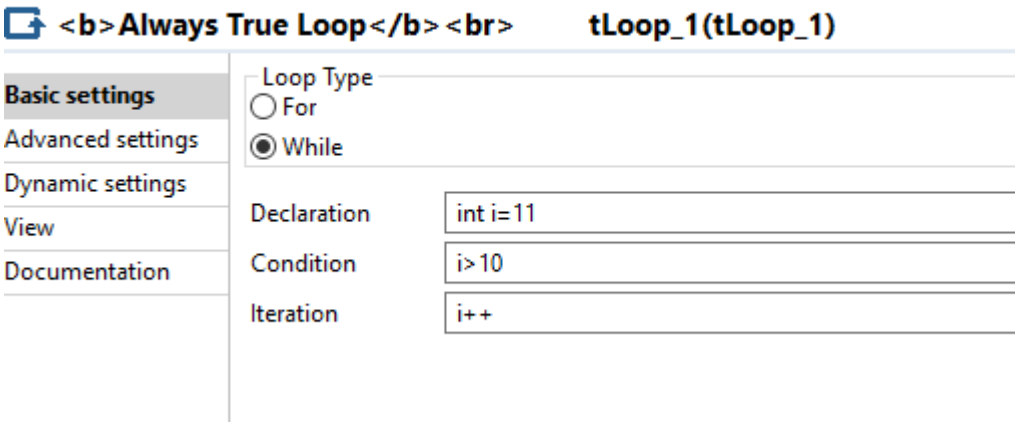


The Access key identifier can be retrieved by viewing the "**My Security Credentials**" option for your account in the AWS console.



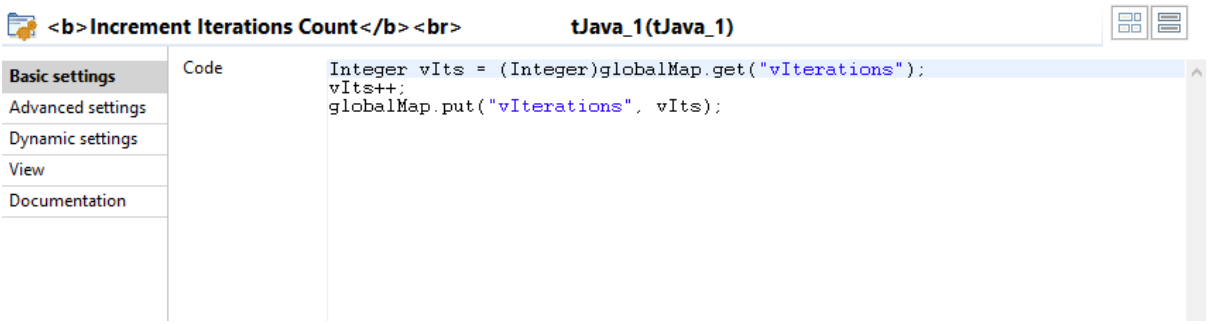
The secret key will be in the file produced when the key pair were generated and should have been stored in a safe place. Select the region that matches the SQS queue location and label the component "**AWS SQS Connection**".

Unlike the JMS and MOM components for queue monitoring, the SQS Input version does not have an option to keep monitoring the queue and will terminate after use. To provide monitoring functionality a **tLoop** is used with a perpetual condition to constantly run the job unless manually terminated.



A **While** loop is selected and arbitrary conditions applied such that it was always equate to **true**, causing the loop to continue indefinitely. A label "**Always True Loop**" should be assigned to the component.

The **tJava** component is called on each iteration of the loop to increment the counter in the global map,

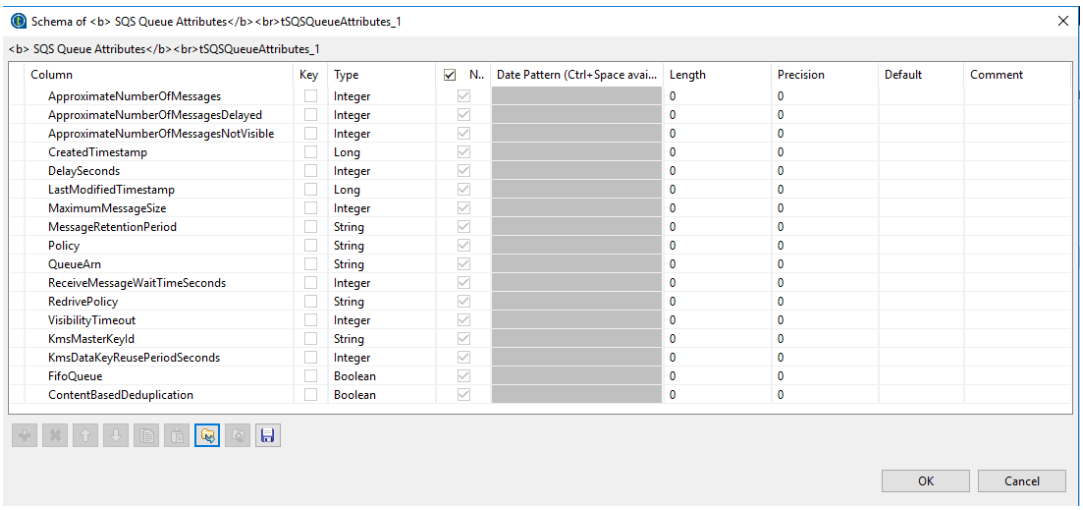


The java snippet retrieves the value of the global variable **vIterations** as an integer, increments it and sends the new value back to the global map. Label the component "**Increment Iterations Count**".

tSQSQueueAttributes allows a peek at the SQS queue status allowing the program to decide whether to apply message retrieval logic for the current iteration of the loop or not. A lot of information about the queue is returned but the item of interest is the **number of messages**.

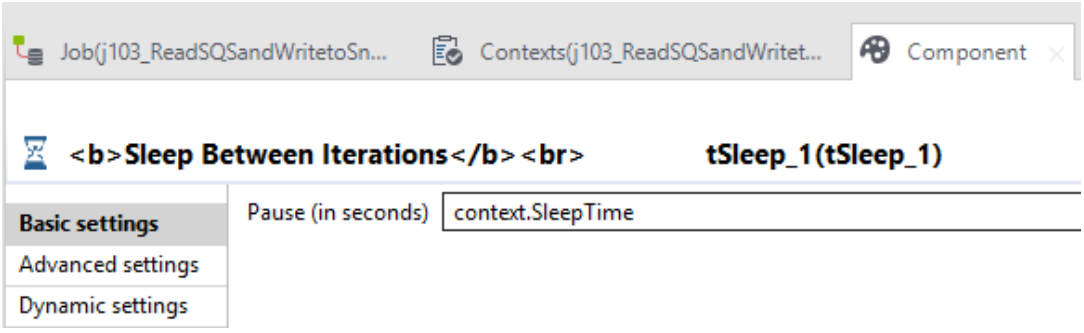
Check the "**Use an existing connection**" option and select the connection component from the list. Add the queue name which can be retrieved from the AWS SQS console.

Make sure to place the value in double quotes as it is a string, otherwise Talend will treat it as a variable and cause an error. Clicking the **Edit schema** ellipsis will show the information returned from SQS.



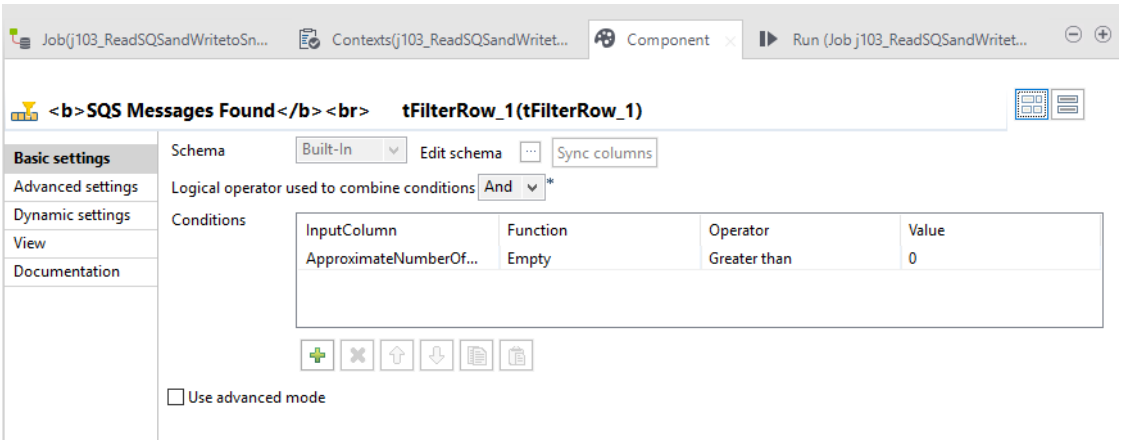
The first item "**ApproximateNumberOfMessages**" is the metric that is of interest to this procedure. Label the component "**SQS Queue Attributes**".

The **tSleep** component will introduce a pause into the loop iteration controlled by the value of the context variable defined in the context file.



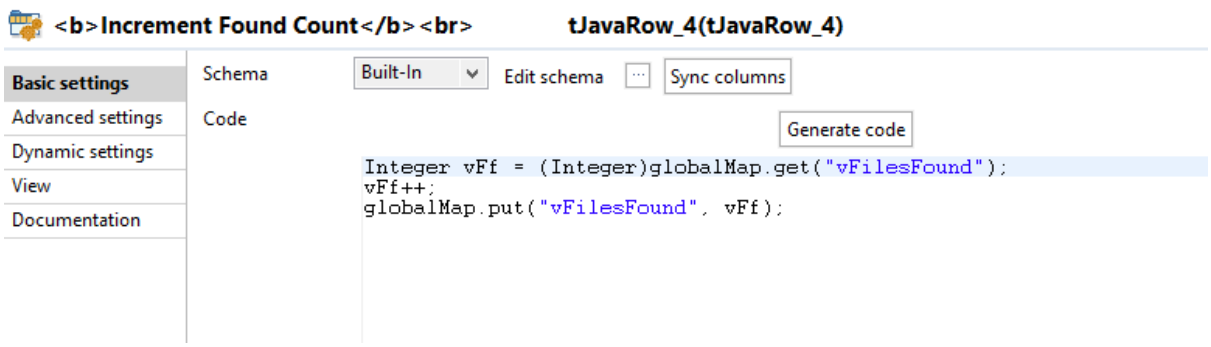
Attached by an OnComponentOK it will be fired every iteration of the loop. Label the component "Sleep Between Iterations"

A **tFilterRow** is used to control the program flow based on the SQS status data returned.



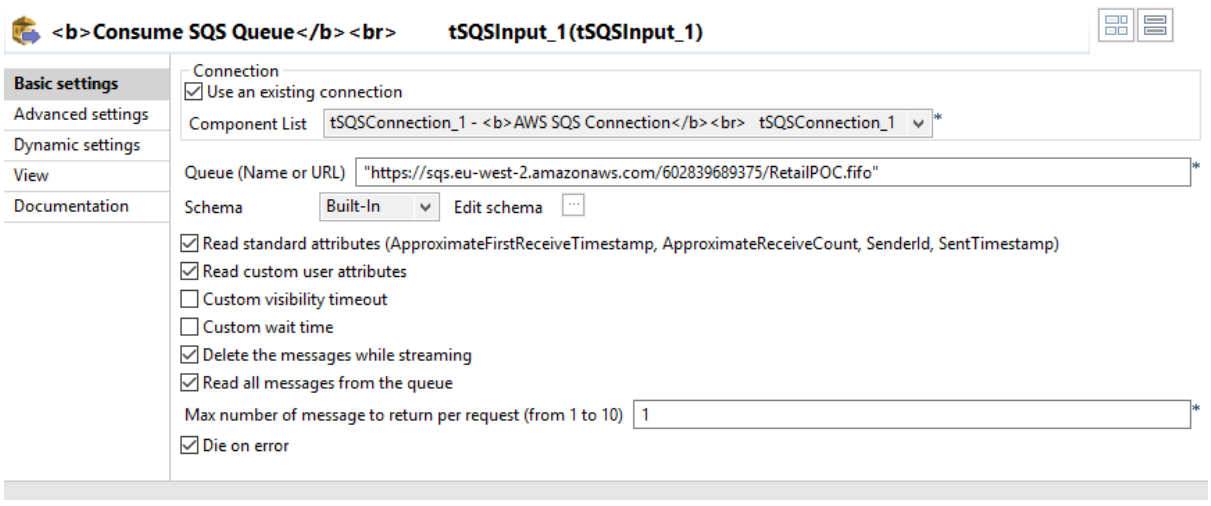
Click the **green + icon** to add a filter row and select the "**ApproximateNumberOfMessages**" option from the InputColumn drop-down list. Leave the Function empty and select "**Greater than**" as the Operator. Add a Value of 0 to allow the filter when message(s) are present and add the label "**SQS Messages Found**".

A **tJavaRow** component is used to increment the message found count in the global map.



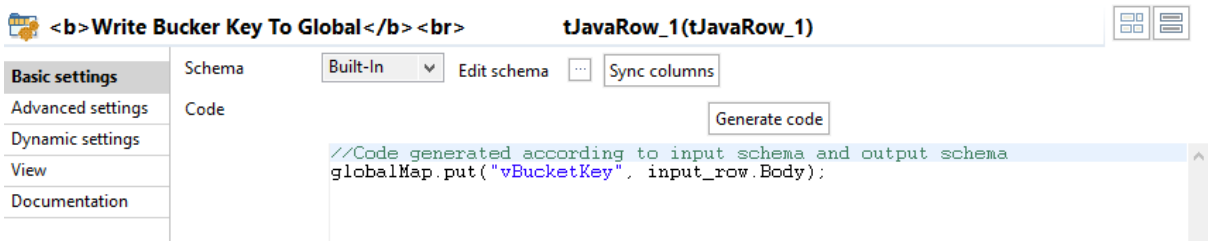
Very similar to the earlier **tJava** component but called via a filer Row rather than a trigger. The java snippet retrieves the value of the global variable **vFilesFound** as an integer, increments it and sends the new value back to the global map. Label the component "**Increment Found Count**".

SQS Messages are consumed by the **tSQSInput** component. It is configured using the same information as the **tSQSQueueAttributes** component.



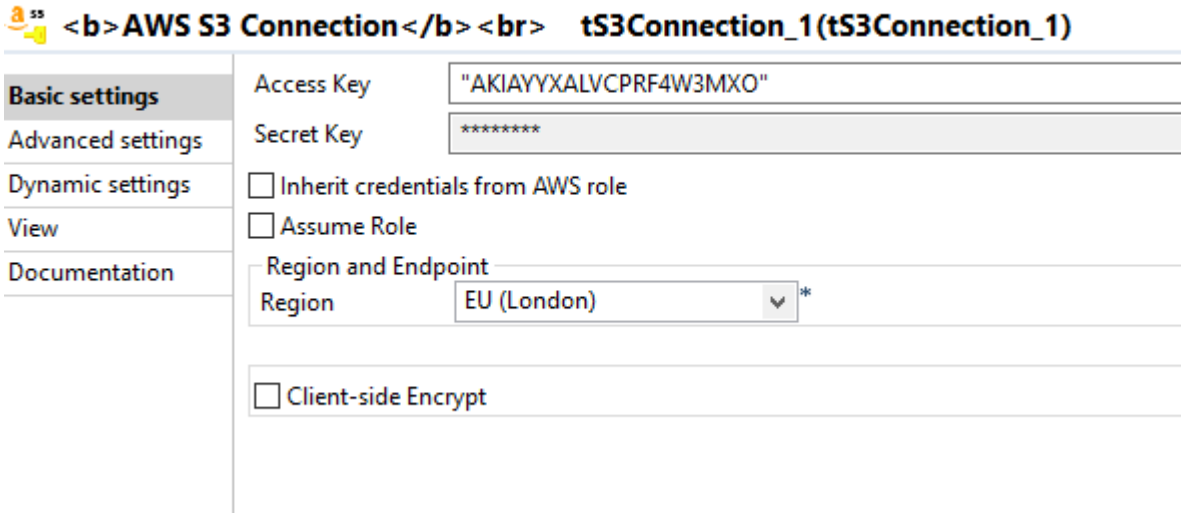
Check the "**Use an existing connection**" option and select the connection component from the list. Add the queue name which can be retrieved from the AWS SQS console as previously shown. Label the component "**Consume SQS Queue**".

The object key for the file in S3 storage is contained in the body of the message, having been placed there by the AWS Lambda function. A second **tJavaRow** component is used to assign the key value to the **vBucketKey** variable in the global map.



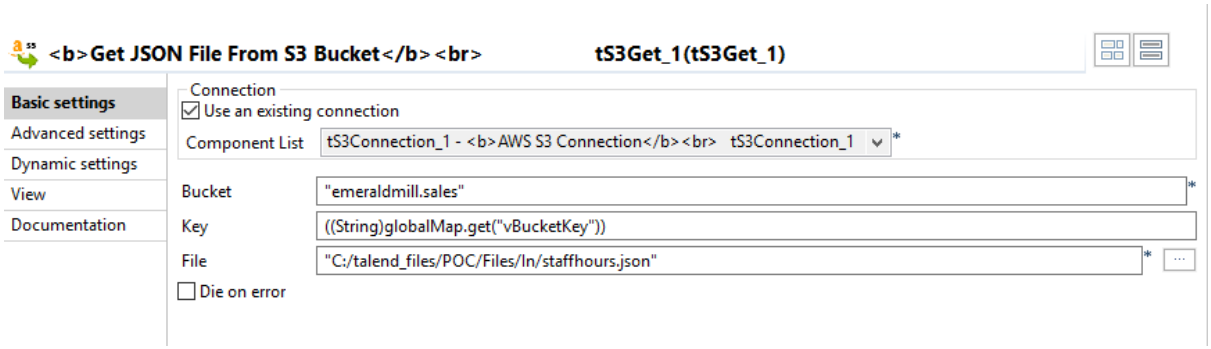
The simple java snippet overwrites the current value with the contents of the message body. Label the component "**Write Bucket Key To Global**".

Connection to the S3 bucket is initiated by the **tS3Connection** component.



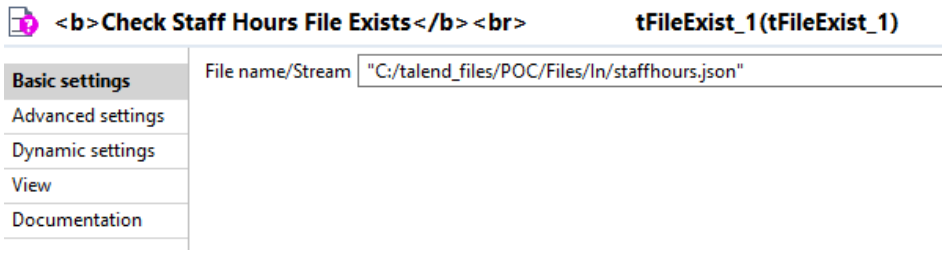
Use the keys generated when creating the bucket, as described earlier in this document and select the AWS Region which should match the bucket location. Label the component "**AWS S3 Connection**"

The **tS3Get** component is used to retrieve the file form the S3 bucket using the global bucket key variable assigned previously.



Check the "**Use an existing connection**" option and select the connection component from the list. Add the bucket name "**emeraldmill.sales**" and retrieve the variable from the global map for the key, **((String)globalMap.get("vBucketKey")).** In the File editor add the full path name of the local file that it will be saved to, "**C:/talend_files/POC/Files/In/staffhours.json**". Label the component "**Get JSON File From S3 Bucket**".

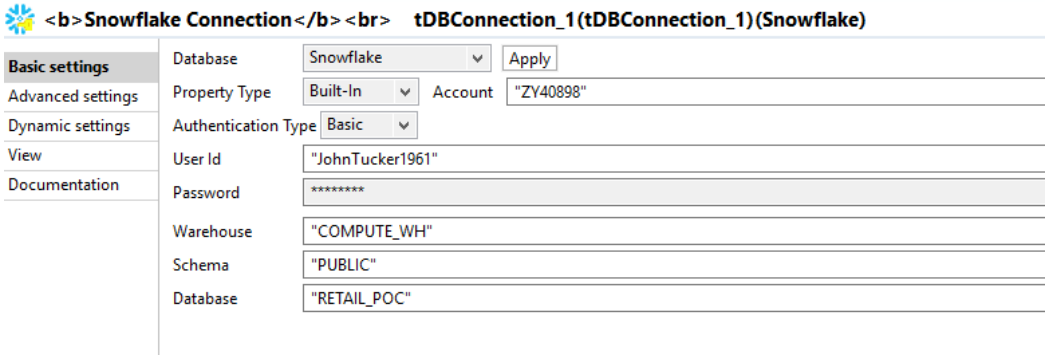
A **tFileExist** component is used to check the success of the previous operation by validating the existence of the local file copy after download from S3.



Add the name of the local file into the **File name/stream** editor and label the component "**Check Staff Hours File Exists**"

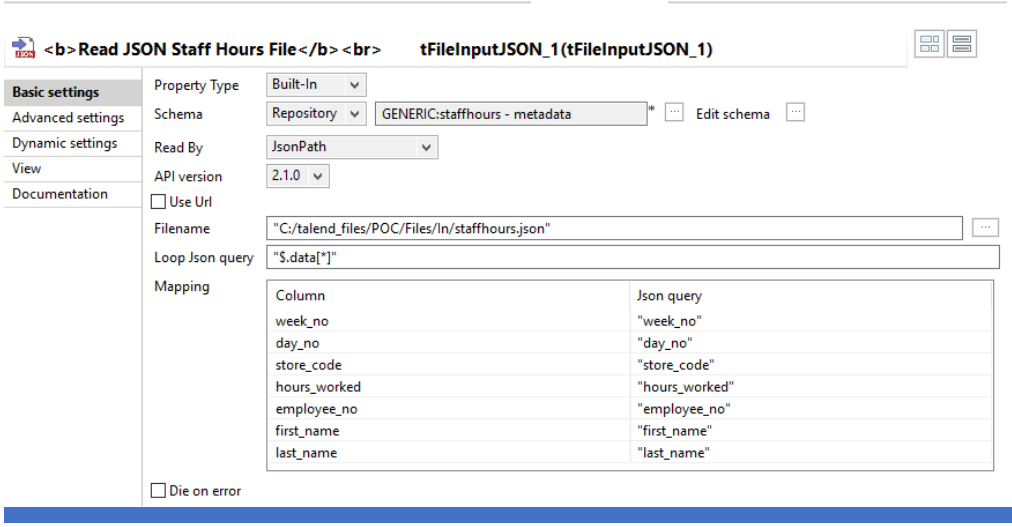
A **tDBConnection** component is configured to access the database set up earlier in this document using the **SnowflakeDB** account.

This proof of concept is set up to use a trial Snowflake account, but it could easily be converted to use a similar service such as AWS Redshift or Google cloud platform (GCP) Big Query. Alternatively, any ANSI compliant RDBMS, cloud based or local could be used.



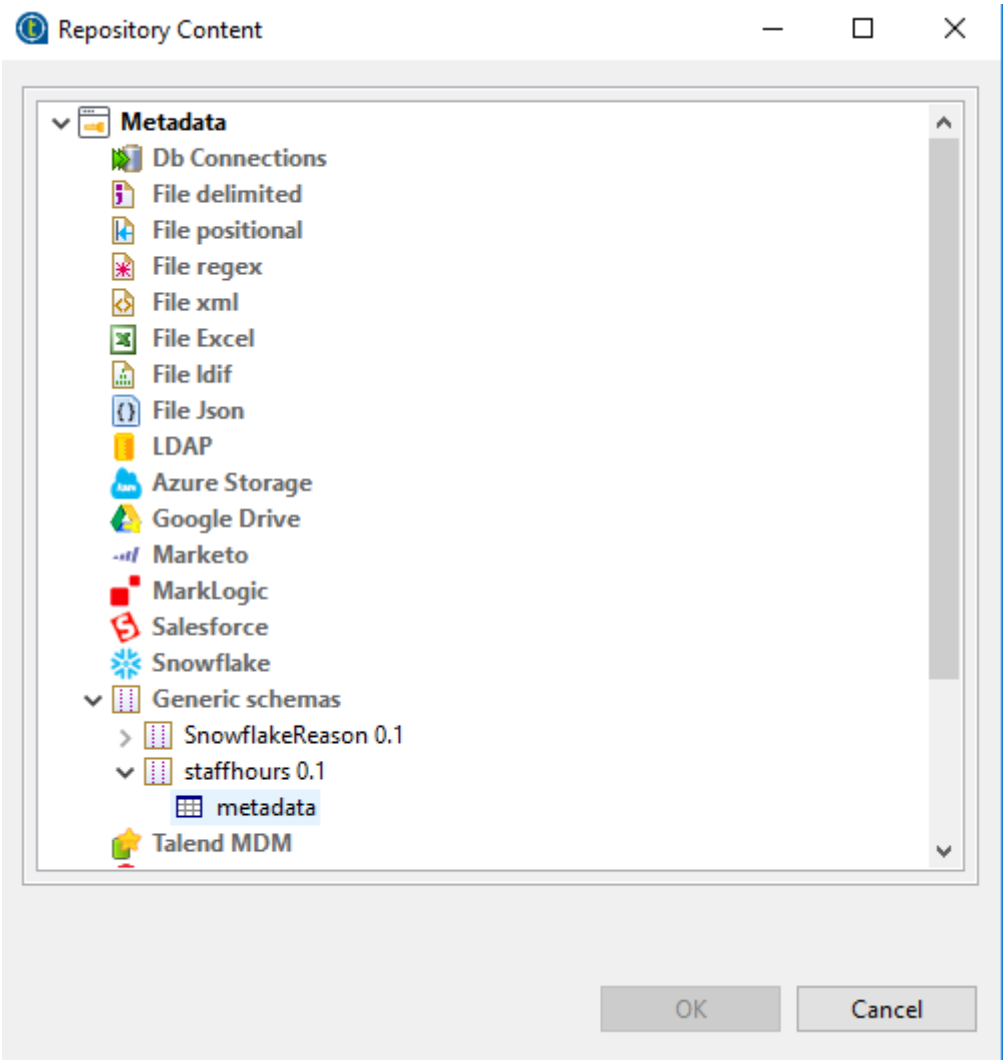
Select **Snowflake** as the database and enter account number together with **User Id** & **Password**. The **Warehouse, Schema** and **Database** fields refer to the Snowflake setup earlier, "**COMPUTE_WH**", "**PUBLIC**" and "**RETAIL_POC**" in this case. Label the component "**Snowflake Connection**".

The local JSON file downloaded from S3 is accessed by the **tFileInputJSON** component.



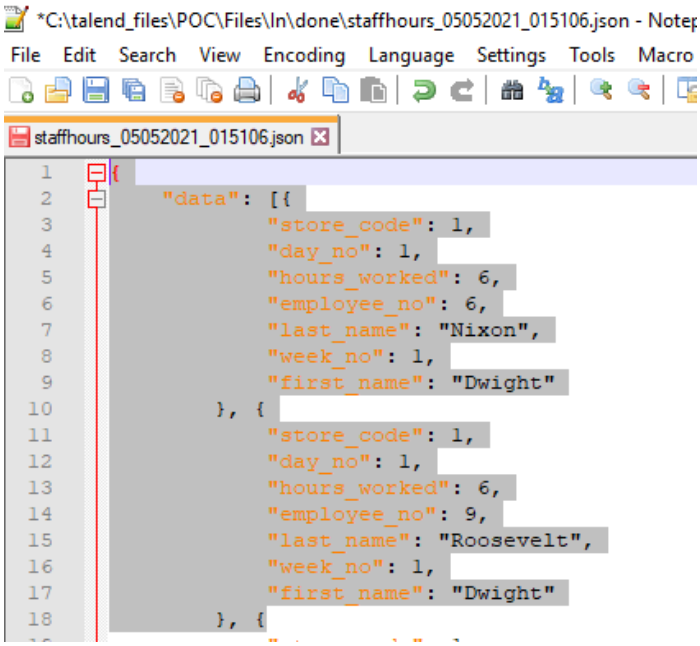
A schema created for the previous job and saved in the repository as "**staffhours**" can be reused as the JSON file contains the same field layout. Select the **Repository** option from the Schema drop-down list which will add an edit control for the repository schema name.

Click *the **ellipsis*** next to the control to call the repository viewer.

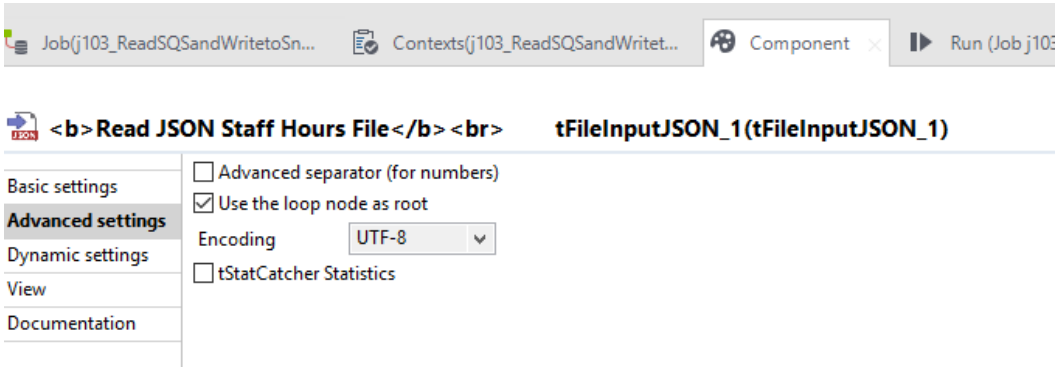


Expand the **Generic schemas** section. Select **staffhours** and click **OK** to complete. You can check the schema by clicking **Edit schema** and selecting the **View Schema** option.

The file will be read using **JsonPath** which is the JSON equivalent of XPath, utilising the latest version of the API 2.1.0. Filename will be the local JSON file path and the loop expression should be **"$.data[*]"** which will retrieve data for each occurrence of the "**data**" node.

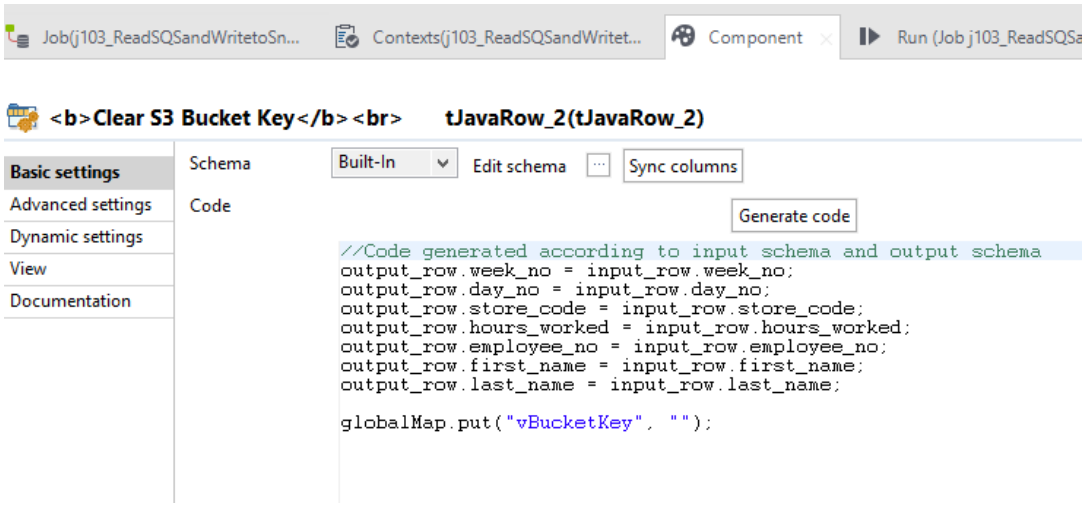


In the mapping section the values in the Json query column should match the previous column but be enclosed in double quotes.



Ensure that the "**Use the loop node as root**" is checked in the **Advanced settings** section then label the component "**Read JSON Staff Hours File**".

A third **tJavaRow** component is used to clear the value of the bucket key global variable.



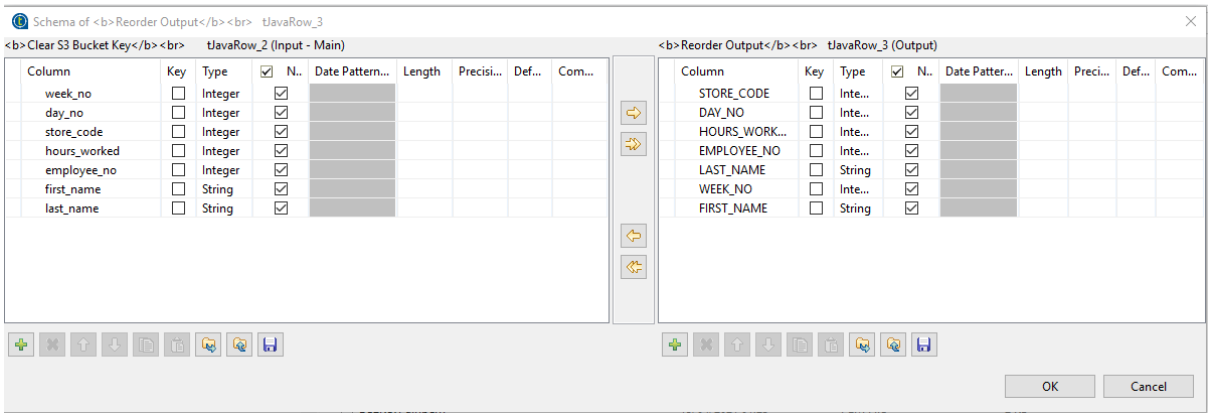
To ensure the data flow passes through the component click the **Sync columns** button followed by **Generate code**. This will generate code assigning each input value to a corresponding output. Failure to complete these steps would mean the input data was lost to the chain downstream from this component.

Always do the previous steps first prior to any alterations otherwise Generate code will delete any work you have done in the editor and replace with the input output assignment.

Add this line below the generated code to clear the variable "**globalMap.put("vBucketKey", "")**;" and label the component "**Clear S3 Bucket Key**".

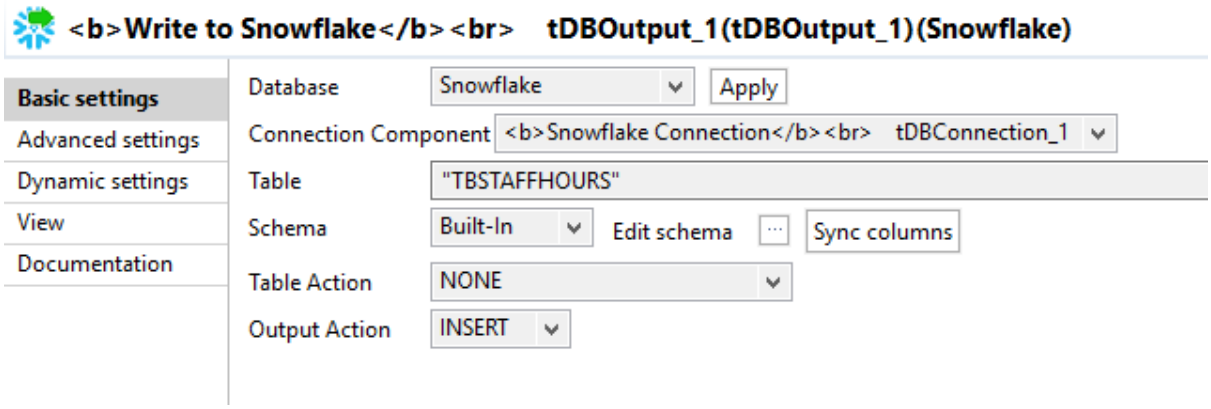
A fourth **tJavaRow** component is used to re-order the output into the format requires by **Snowflake**.

Click on the **Edit schema** ellipsis to call the editor.



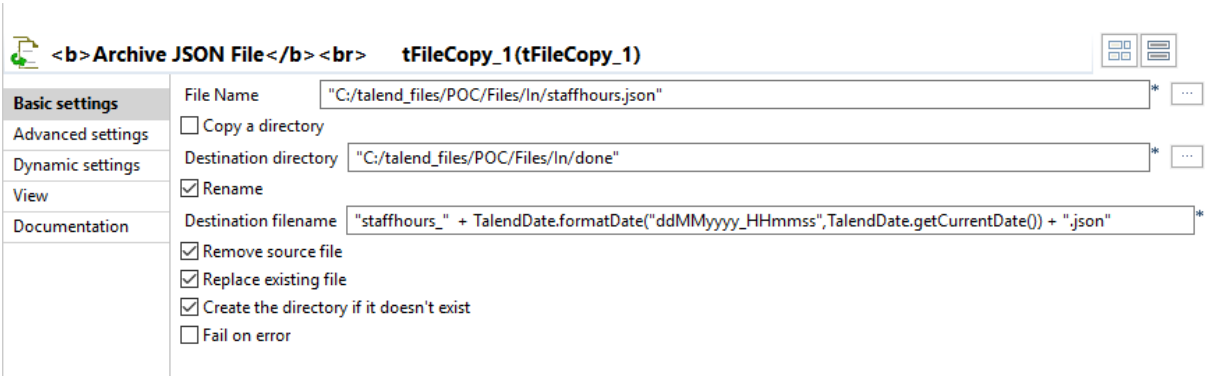
Modify the output schema by changing the column name to **upper case** and changing the order as shown in the diagram above. Click **OK** to close the editor then **Generate code**. The correct output should be produced. Label the component "**Reorder Output**".

Writing the data to the SnowflakeDB table is handled by a **tDBOutput** component.



Click **Sync columns** to incorporate any changes made to the previous component. Select **Snowflake** as the database and the Snowflake connection component from the list. Set the table name to "**TBSTAFFHOURS**" with the Output Action of "**INSERT"**. Label the component "**Write to Snowflake**".

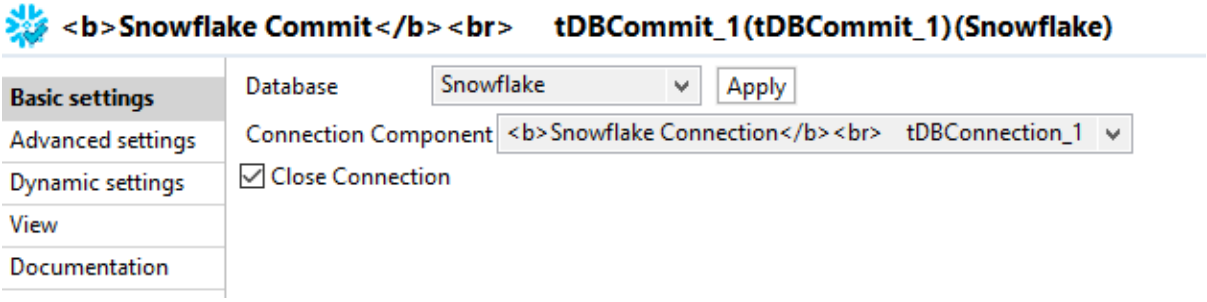
A **tFileCopy** component is used to archive the local JSON file after use.



Enter the full path of JSON file "**C:/talend_files/POC/Files/In/staffhours.json**" and the destination directory "**C:/talend_files/POC/Files/In/done**". Check the **Rename** option and add the following expression for the Destination filename "**staffhours_" + TalendDate.formatDate("ddMMyyyy_HHmmss",TalendDate.getCurrentDate()) + ".json**".

Check the **Remove source file, Replace existing file** and **Create the directory if it doesn't exist** options and label the component "**Archive JSON File**".

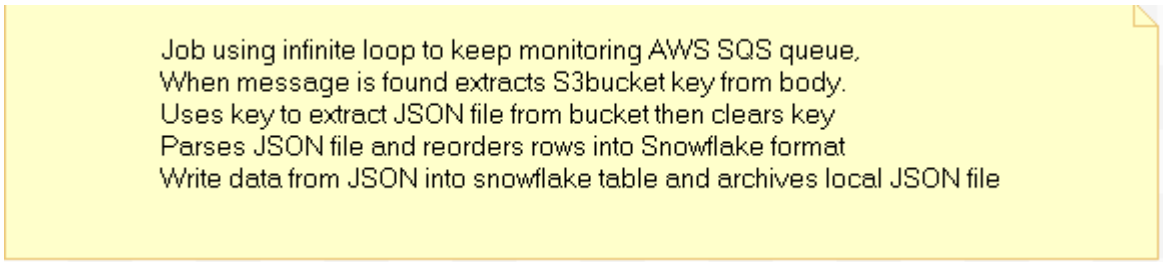
The final component is **tDBCommit** to commit Snowflake changes to the database.



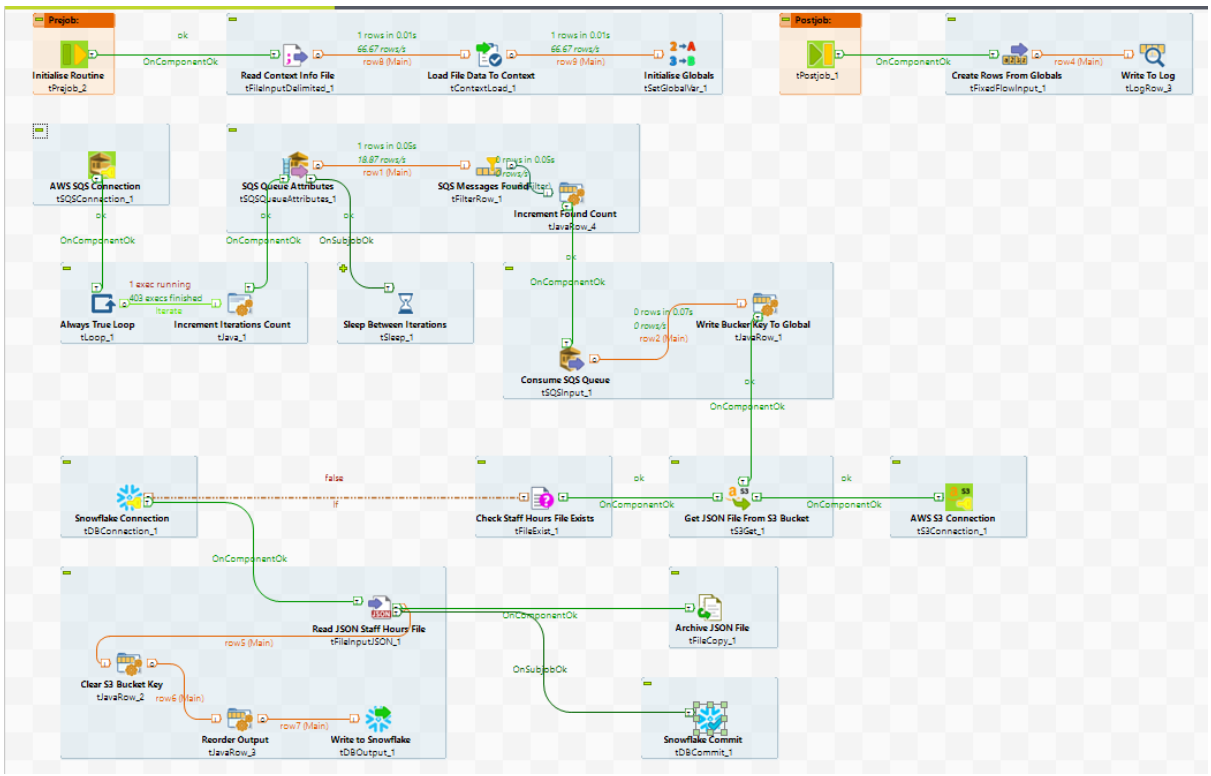
Select **Snowflake** as the database and the DB connection from the list. Check the **Close Connection** box.

As the commit operation is initiated by an OnSubjobOk trigger from the routine that includes the Snowflake output component, completion of the database write operation is guaranteed prior to the commit. It is therefore safe to close the connection as part of the commit.

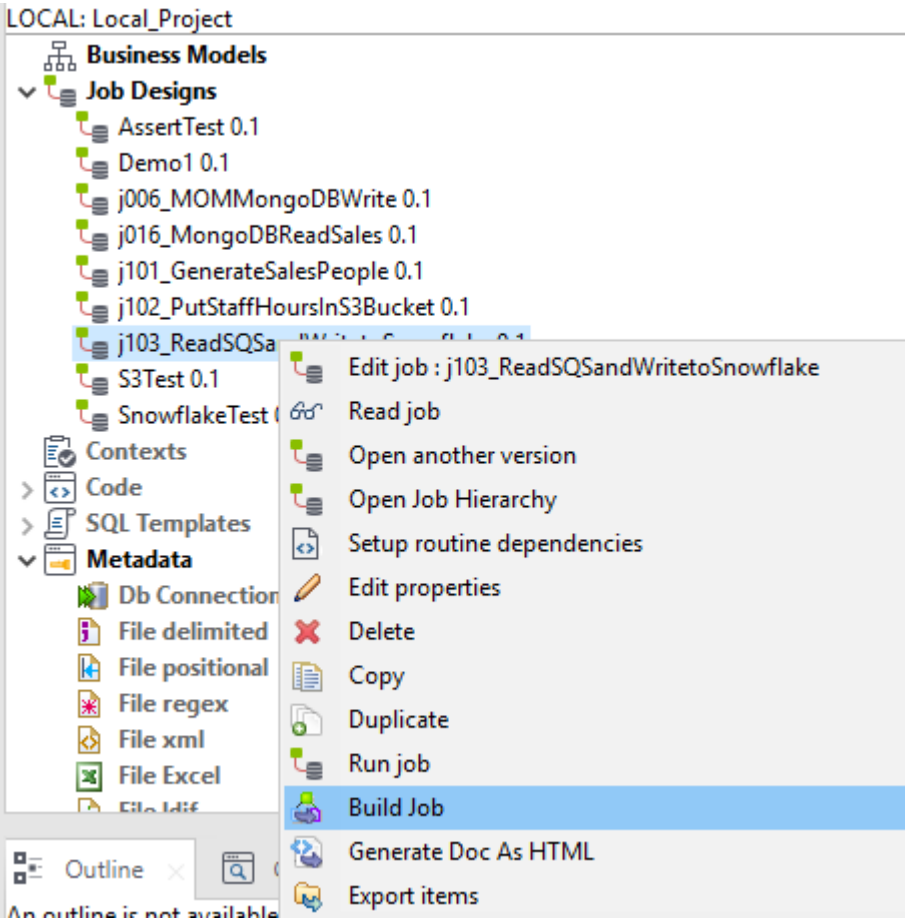
Label the component as "**Snowflake Commit**" and this completes the configuration of the components. For documentation purposes add the following **note** to the design area.



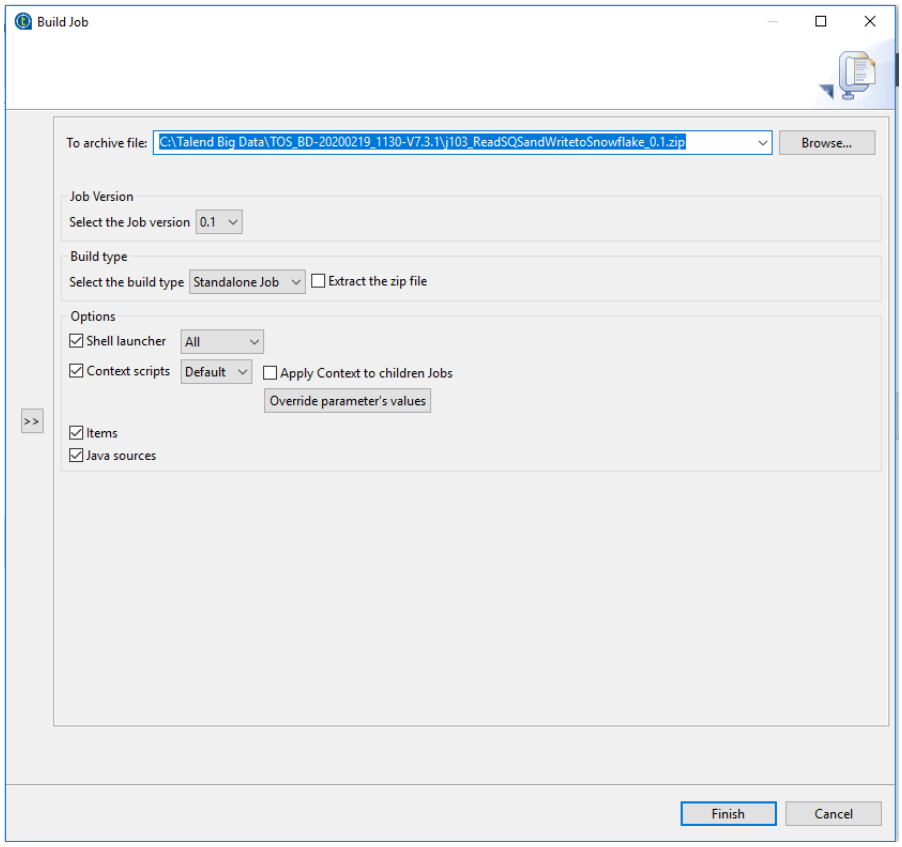
The completed job should resemble the illustration below.



Build the job as **stand-alone** to run on the EC2 instance. Begin the build by saving the job then right click on its name under **Job Designs** in the repository pane and select **Build Job**.



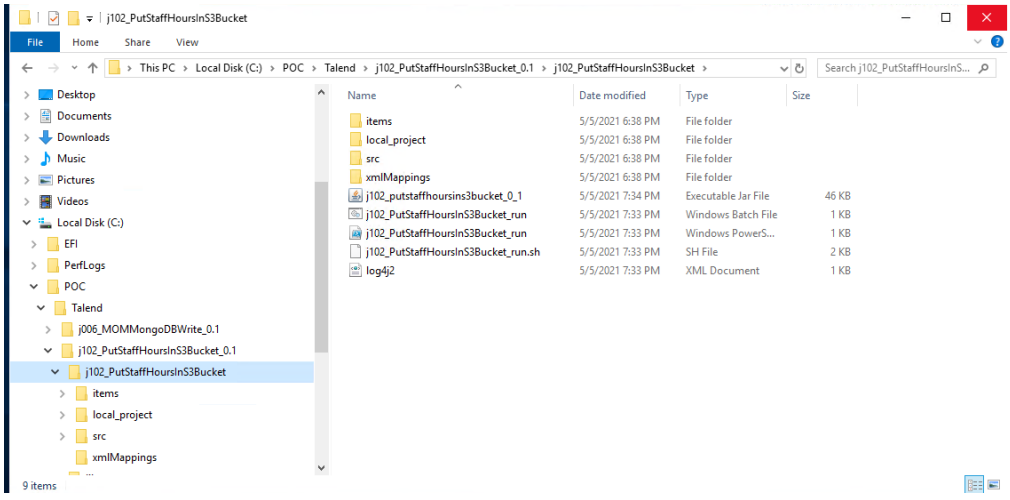
Accept all default values in the build screen and click **Finish** to complete.



Once complete a zip file will be created in the location specified. In this case that is the root folder of Talend Studio although that can be changed if required.

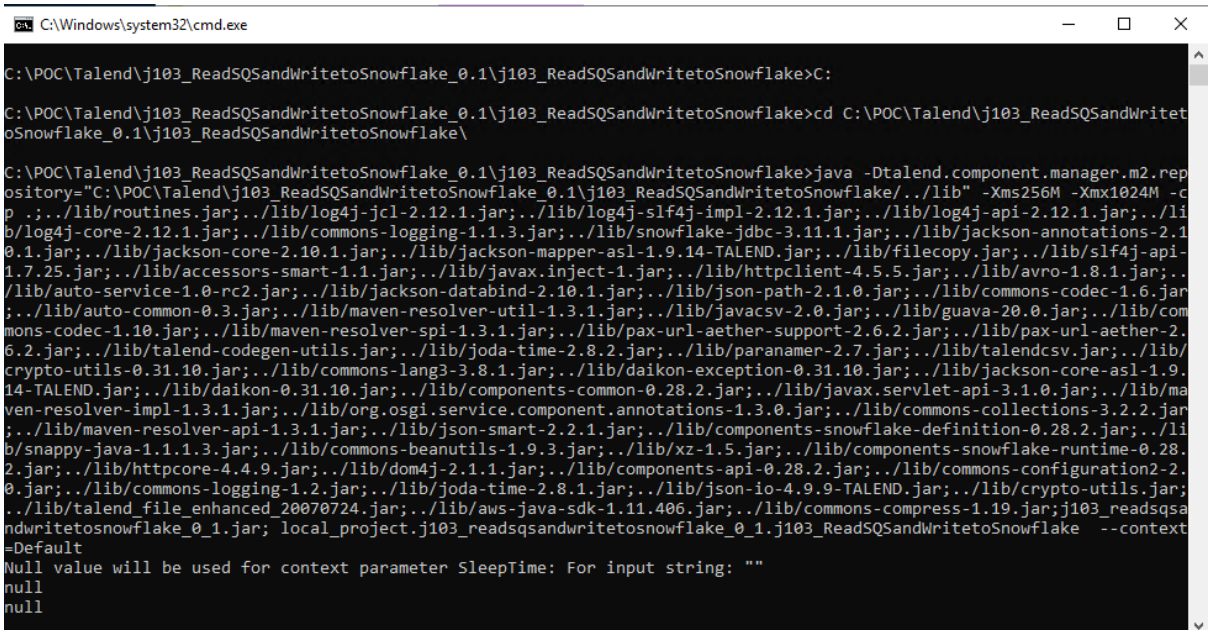
14. Deploy jobs to EC2

The deployment process is identical for both Talend jobs. Copy the entire zip files produced by the build process to the **EC2** server and extract the contents to the desired location. Once extracted look in the folder structures for the windows batch files **j102_PutStaffHoursInS3Bucket_run.bat** and **j103_ReadSQSandWritetoSnowflake_run.bat** .

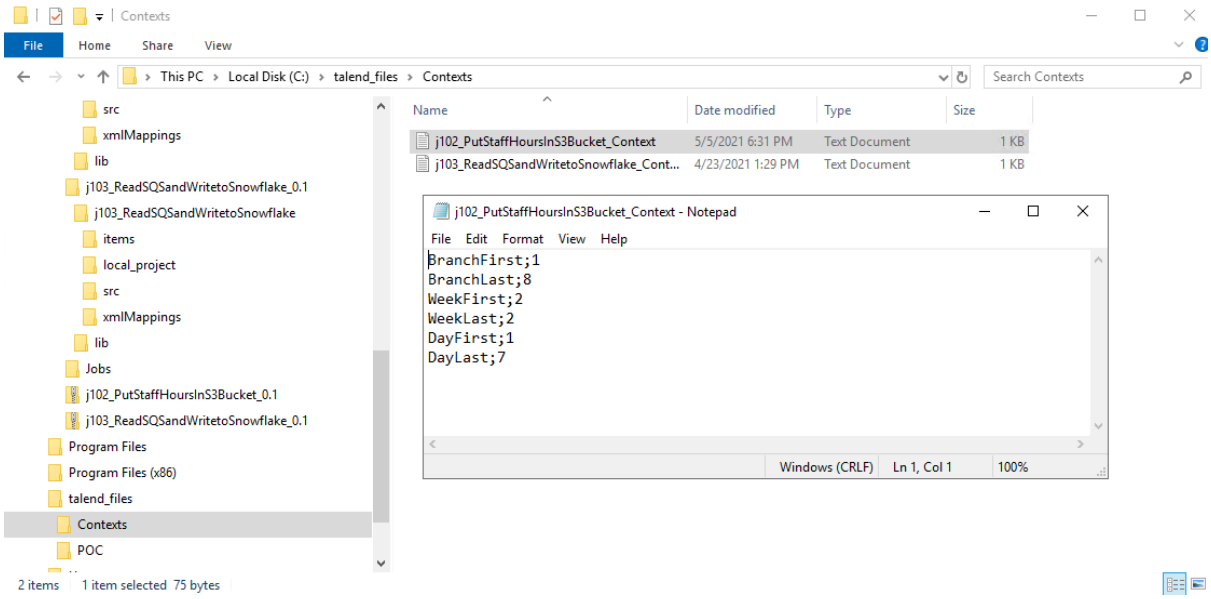


Clicking on either batch file will run the respective job although they could also be running via **Task Scheduler** or if the subscription product was used, be submitted by **Talend Job Server**.

Store staff records have already been generated. To run the rest of the process, double-click the **j103_ReadSQSandWritetoSnowflake_run.bat** file which will start the SQS queue monitoring job in a loop. Once initiated this will remain active until manually terminated assuming no errors intervene.



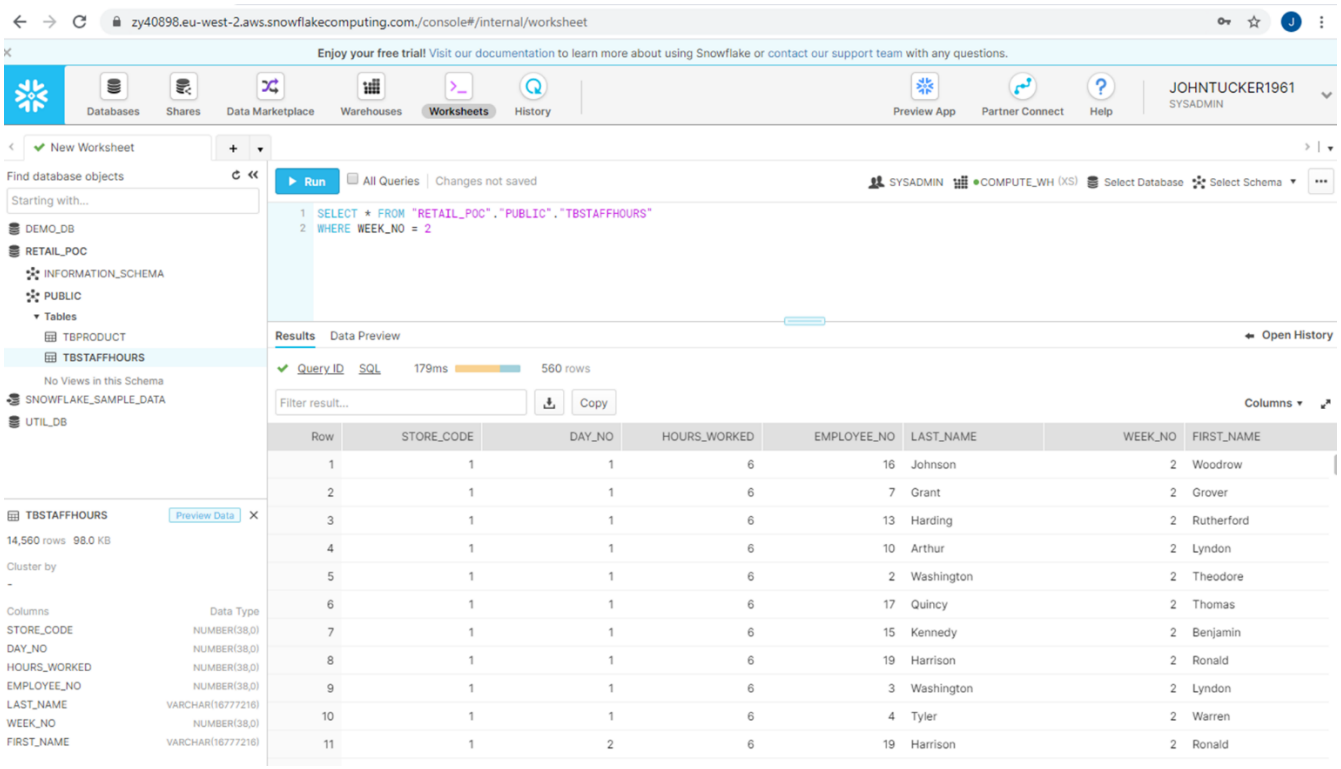
Check the context file for the Talend job to generate time sheet data.



Timesheets for all stores for week 2 should be generated. Click the **j102_PutStaffHoursInS3Bucket_run.bat** file to initiate the process which will launch a separate command window which closes on completion.

The timesheet data is generated in Talend and sent as a **JSON** format file to an **S3** bucket. Arrival in the bucket triggers an AWS **Lambda f**unction which writes the file name to an **SQS** message. The

other Talend job is monitoring the **SQS** queue and on receipt of a new message uses it to retrieve the **JSON** file from **S3**. The data is then parsed and written to **SnowflakeDB**.



That completes this tutorial. In the real world the data would have come from multiple sources rather than being simulated by Talend but the purpose of this proof of concept was to demonstrate the connectivity capabilities of the product using modern cloud-based technologies.