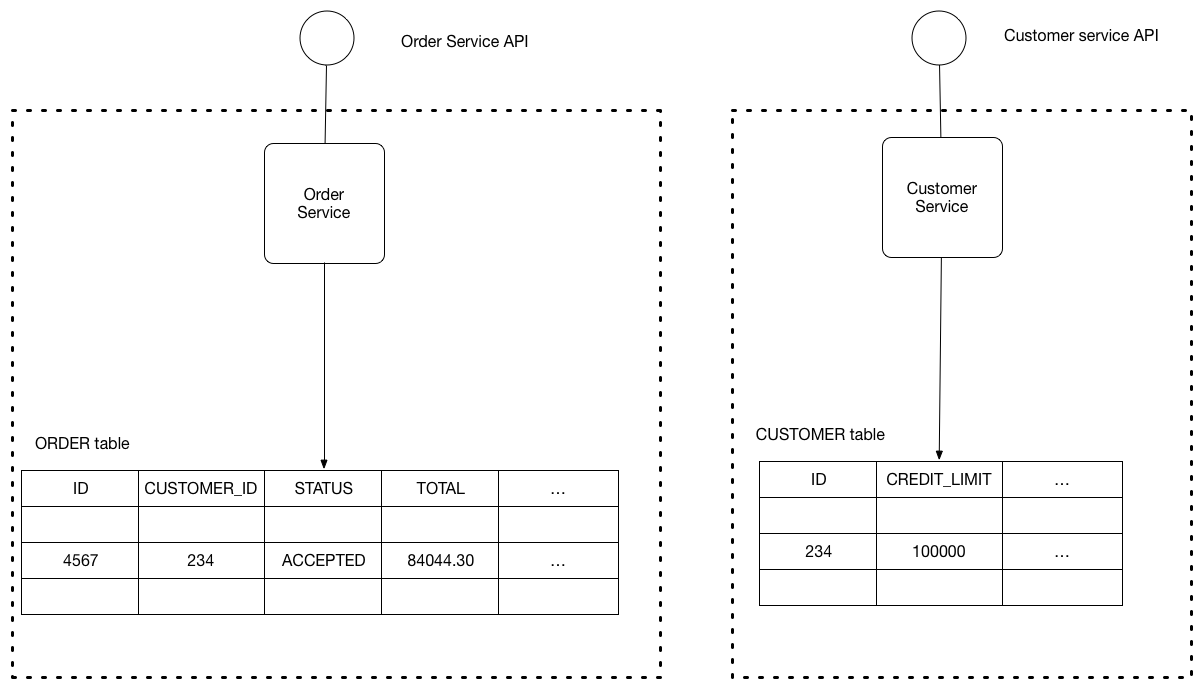
***Event-Driven Architecture and Change Data Capture***

An event is both notification and state transfer. It could be anything from a sensor reading, database changes (both schematic and data changes), customer ratings and a way to communicate between micro services as far as the developers are concerned.

Before we deep dive in the actual concepts and the “[topic](https://www.javatpoint.com/kafka-topics)”, let me give you the basic idea of event driven architecture and changed data capture is extensively used in the so called “[Saga Pattern for Micro Services](https://microservices.io/patterns/data/saga.html)”. The need of Saga Pattern came into picture due to the usage of “[Database per Service Pattern](https://microservices.io/patterns/data/database-per-service.html)”. The Database per Service Pattern, as the name suggests is the pattern that has one dedicated database for a service (in some cases it is even one or more dedicated tables for a service). This pattern is used to meet the requirements for distributed systems and architecture of micro services. And apart from these patterns, “[CQRS Pattern](https://microservices.io/patterns/data/cqrs.html)” is also used to segregate the reading and writing logics in the system.

But we will talk about all these patterns in details some other day. For now I am just trying to give you the gist so the article makes more sense and you might be able to answer the questions like WHY/HOW DOES IT MATTER?, WHY SHOULD I BOTHER?, WHY THAT MUCH PAIN?, AND WHY CAN’T THE WORLD LIVE PEACEFULLY? :-D



If we talk in the context of an Online Ordering Application or E-Commerce Website, according to the Database per Service Pattern it is advisable to have dedicated databases (or tables) for the Order Service and the Customer Service. The Order Service stores information about orders and the Customer Service stores information about customers as shown in the above figure.  
  
Many challenging scenarios pop up while implementing this Distributed Architecture, like,

* Implementing business transactions that span multiple services is not straightforward. Distributed transactions are best avoided because of the [CAP Theorem](https://www.geeksforgeeks.org/the-cap-theorem-in-dbms/). Moreover, many modern (NoSQL) databases don’t support them.
* Implementing queries that join data which is now in multiple databases is challenging.
* Complexity of managing multiple SQL and NoSQL databases.

To solve such scenarios, the concepts of Saga Pattern (the solution of the scenario depicted in the figure is [here](https://microservices.io/patterns/data/saga.html)), Event Driven Architecture and Change Data Capture come into picture.

Now too much of here and there, coming back to the “topic” of this article. Streaming Data Events is the major part of Event Driven Architecture if your system involves multiple databases or services.  
  
[Event stream processing (ESP)](https://hazelcast.com/glossary/event-stream-processing/) is the practice of taking action on a series of data points that originate from a system that continuously creates data. The term “event” refers to each data point in the system, and “stream” refers to the ongoing delivery of those events. A series of events can also be referred to as “streaming data” or “data streams.”

Use cases such as [payment processing](https://hazelcast.com/use-cases/payment-processing/), [fraud detection](https://hazelcast.com/use-cases/fraud-detection/), anomaly detection, predictive maintenance, and [IOT analytics](https://hazelcast.com/use-cases/internet-of-things/) all rely on immediate action on data. All of these use cases deal with data points in a continuous stream, each associated with a specific point in time.

To achieve the above mentioned use cases we have to capture database events as part of Event Streaming. This is where CDC (Change Data Capture) comes into the picture.

***CDC with DEBEZIUM***

[Debezium](https://debezium.io/) is an open source distributed platform for change data capture. Start it up, point it at your databases, and your apps can start responding to all of the inserts, updates, and deletes that other apps commit to your databases. Debezium is durable and fast, so your apps can respond quickly and never miss an event, even when things go wrong.

Fascinating!!! Isn’t it?

I found it amazing as it uses Kafka Connect internally and provides records with detailed information about the database changes. People might argue for JDBC connector as well, but Debezium Connector is way better as far as the Data Streaming is concerned. [You can find the difference between the two here](https://stackoverflow.com/questions/60908891/kafka-connect-jdbc-vs-debezium-cdc).

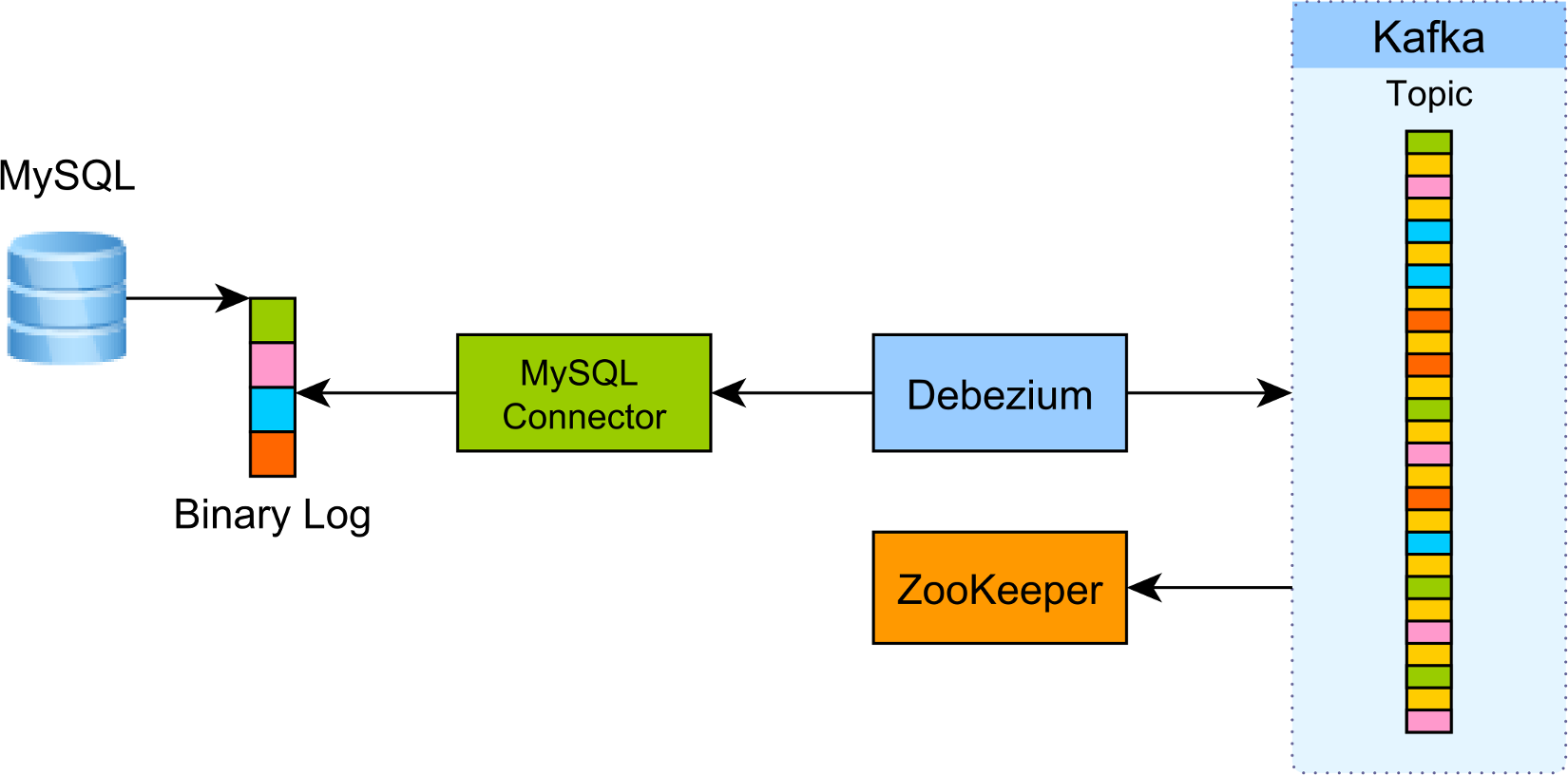
To understand it better let’s play with Debezium and create POCs and verify what its claims.  
To go ahead with POCs, I would be showing 3 different implementations with 3 different databases. I would be only targeting RDBMS for now to make it simple. We will surely explore NoSQL databases some other day.

**LET’S GET STARTED!!!**

**1st POC: Capturing Data from a MySQL DB into the Kafka Cluster + DOCKER. This is pure vanilla setup involving MySQL, Debezium, Kafka and Docker.**

We need a database-specific Debezium connector to be able to extract the Binary Logs of the table. In this case we would need MySQL Connector.

**NOTE**: [Make sure Binary Log is enabled](https://documentation.commvault.com/commvault/v11/article?p=34667.htm).



You also need to have Kafka running so that you can push the extracted log events and make them available to other services in your enterprise system. Apache Zookeeper is not needed by Debezium, but by Kafka since it relies on it for consensus as well as linearity is guaranteed.  
  
Basically, we have to run the following commands to create Docker containers:

* **docker run -it --name zookeeper -p 2181:2181 -p 2888:2888 -p 3888:3888 debezium/zookeeper:0.5**
* **docker run -it --name kafka -p 9092:9092 --link zookeeper:zookeeper debezium/kafka:0.5**
* **docker run -it --name mysql -p 3306:3306 -e MYSQL\_ROOT\_PASSWORD=debezium -e MYSQL\_USER=mysqluser -e MYSQL\_PASSWORD=mysqlpw debezium/example-mysql:0.5**
* **docker run -it --name kafka-connect -p 8083:8083 -e GROUP\_ID=1 -e CONFIG\_STORAGE\_TOPIC=my\_connect\_configs -e OFFSET\_STORAGE\_TOPIC=my\_connect\_offsets --link zookeeper:zookeeper --link kafka:kafka --link mysql:mysql debezium/connect:0.5**
* **docker run -it --name kafka-watcher --link zookeeper:zookeeper debezium/kafka:0.5 watch-topic -a -k dbserver1.inventory.customers**

These commands would create following containers:

* **docker ps -a**

**CONTAINER ID        IMAGE                          NAMES**

**bbfeafd9125c        debezium/kafka:0.5             kafka-watcher**

**4cfffedae69c        debezium/connect:0.5           kafka-connect**

**36734bc82864        debezium/example-mysql:0.5     mysql**

**daaaab6f3206        debezium/kafka:0.5             kafka**

**8a7affd3e2a4        debezium/zookeeper:0.5         zookeeper**

Using bash, we would need to create a new connector. Run the below command:

* **curl -i -X POST -H "Accept:application/json" -H "Content-Type:application/json" localhost:8083/connectors/ -d '{ "name": "inventory-connector", "config": { "connector.class": "io.debezium.connector.mysql.MySqlConnector", "tasks.max": "1", "database.hostname": "mysql", "database.port": "3306", "database.user": "debezium", "database.password": "dbz", "database.server.id": "184054", "database.server.name": "dbserver1", "database.whitelist": "inventory", "database.history.kafka.bootstrap.servers": "kafka:9092", "database.history.kafka.topic": "dbhistory.inventory" }}'**

Notice that kafka-watcher was started in interactive mode so that we can see in the console the CDC log events captured by Debezium.

## Test!! If it works…

Now, if we connect to the MySQL Docker container using the root user and the debezium password, we can issue various SQL statements and inspect the kafka-watcher container console output.

INSERT

When inserting a new customer row:

|  |  |
| --- | --- |
|  | **INSERT INTO `inventory`.`customers`**  **(**  **`first\_name`,**  **`last\_name`,**  **`email`)**  **VALUES**  **(**  **'Divyesh',**  **'Bhartiya',**  **'Divyesh@acme.org'**  **)** |

In the kafka-watcher, we can now find the following JSON entry:

|  |  |
| --- | --- |
|  | **{**  **"payload":{**  **"before":null,**  **"after":{**  **"id":1005,**  **"first\_name":"Divyesh",**  **"last\_name":"Bhartiya",**  **"email":"Divyesh@acme.org"**  **},**  **"source":{**  **"name":"dbserver1",**  **"server\_id":223344,**  **"ts\_sec":1500369632,**  **"gtid":null,**  **"file":"mysql-bin.000003",**  **"pos":364,**  **"row":0,**  **"snapshot":null,**  **"thread":13,**  **"db":"inventory",**  **"table":"customers"**  **},**  **"op":"c",**  **"ts\_ms":1500369632095**  **}**  **}** |

The before object is null while the after object shows the newly inserted value. Notice that the op attribute value is c, meaning it’s a CREATE event.

UPDATE

When updating the customer row:

|  |  |
| --- | --- |
|  | **UPDATE `inventory`.`customers`**  **SET**  **`email` = 'Divyesh.Bhartiya@acme.org'**  **WHERE**  **`id` = 1005;** |

We can now find the following log event:

|  |  |
| --- | --- |
|  | **{**  **"payload":{**  **"before":{**  **"id":1005,**  **"first\_name":"Divyesh",**  **"last\_name":"Bhartiya",**  **"email":"Divyesh@acme.org"**  **},**  **"after":{**  **"id":1005,**  **"first\_name":"Divyesh",**  **"last\_name":"Bhartiya",**  **"email":"Divyesh.Bhartiya@acme.org"**  **},**  **"source":{**  **"name":"dbserver1",**  **"server\_id":223344,**  **"ts\_sec":1500369929,**  **"gtid":null,**  **"file":"mysql-bin.000003",**  **"pos":673,**  **"row":0,**  **"snapshot":null,**  **"thread":13,**  **"db":"inventory",**  **"table":"customers"**  **},**  **"op":"u",**  **"ts\_ms":1500369929464**  **}**  **}** |

The op attribute value is u, meaning we have an UPDATE log event. The before object shows the row state before the update while the after object captures the current state of the updated customer database row.

DELETE

When issuing a DELETE statement:

|  |  |
| --- | --- |
|  | DELETE FROM `inventory`.`customers`  WHERE id = 1005; |

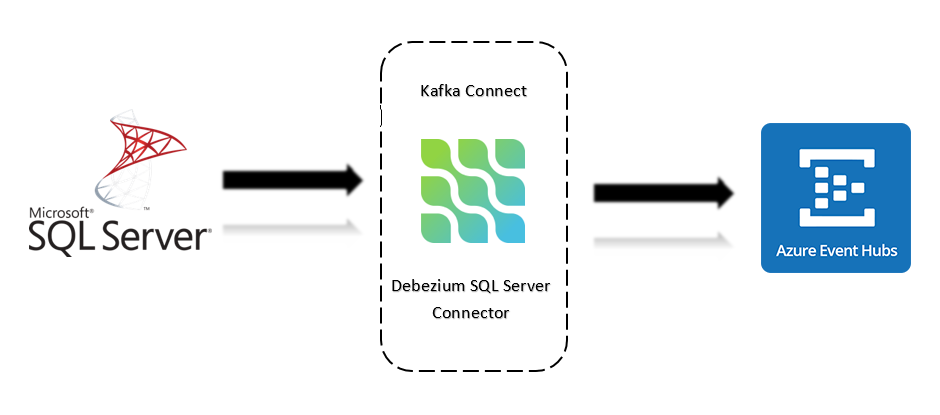
The following event is being recorded by the kafka-connect Docker container:

|  |  |
| --- | --- |
|  | **{**  **"payload":{**  **"before":{**  **"id":1005,**  **"first\_name":"Divyesh",**  **"last\_name":"Bhartiya",**  **"email":"Divyesh.Bhartiya@acme.org"**  **},**  **"after":null,**  **"source":{**  **"name":"dbserver1",**  **"server\_id":223344,**  **"ts\_sec":1500370394,**  **"gtid":null,**  **"file":"mysql-bin.000003",**  **"pos":1025,**  **"row":0,**  **"snapshot":null,**  **"thread":13,**  **"db":"inventory",**  **"table":"customers"**  **},**  **"op":"d",**  **"ts\_ms":1500370394589**  **}**  **}** |

The op attribute value is d, meaning we have a DELETE log event and the after object is now null. The before object captures the database row state before it got deleted.

Brilliant, right??

**2nd POC: Streaming Events from SQL Server to Azure Event Hub using Debezium.**



Streaming SQL Server CDC to Azure Event Hub

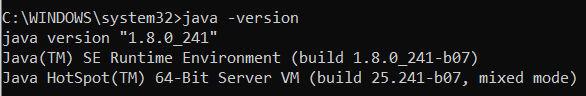
In this part of the article, we will follow the below steps:

* Install Java
* Create Azure Event Hub namespace
* Setup Kafka Connect with Debezium SQL Server Connector
* Setup SQL Server Management Studio and enable CDC (Change Data Capture) on table
* Stream Event from SQL Server to Azure Event Hub

**Java Installation**

Make sure you have java in your machine.

Run the command, **java -version** in your terminal, to check for the same.



**Creating an Event Hub Namespace:**

Follow the above link from Microsoft’s Official Documentation to create an Event Hub Namespace.

<https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-create>

**Setting up Kafka Connect and Debezium:**

What is Kafka Connect?

**Kafka Connect** is a tool for scalable and reliably streaming data between Apache Kafka and other data systems. It makes it simple to quickly define connectors that move large data sets into and out of Kafka.

**What is Debezium?**

Debezium is built on top of [Apache Kafka](http://kafka.apache.org/) and provides [Kafka Connect](https://kafka.apache.org/documentation.html#connect) compatible connectors that monitor specific database management systems. Start it up, point it at your databases, and your apps can start responding to all of the inserts, updates, and deletes that other apps commit to your databases.

[Download Kafka](https://archive.apache.org/dist/kafka/2.6.0/kafka_2.13-2.6.0.tgz)

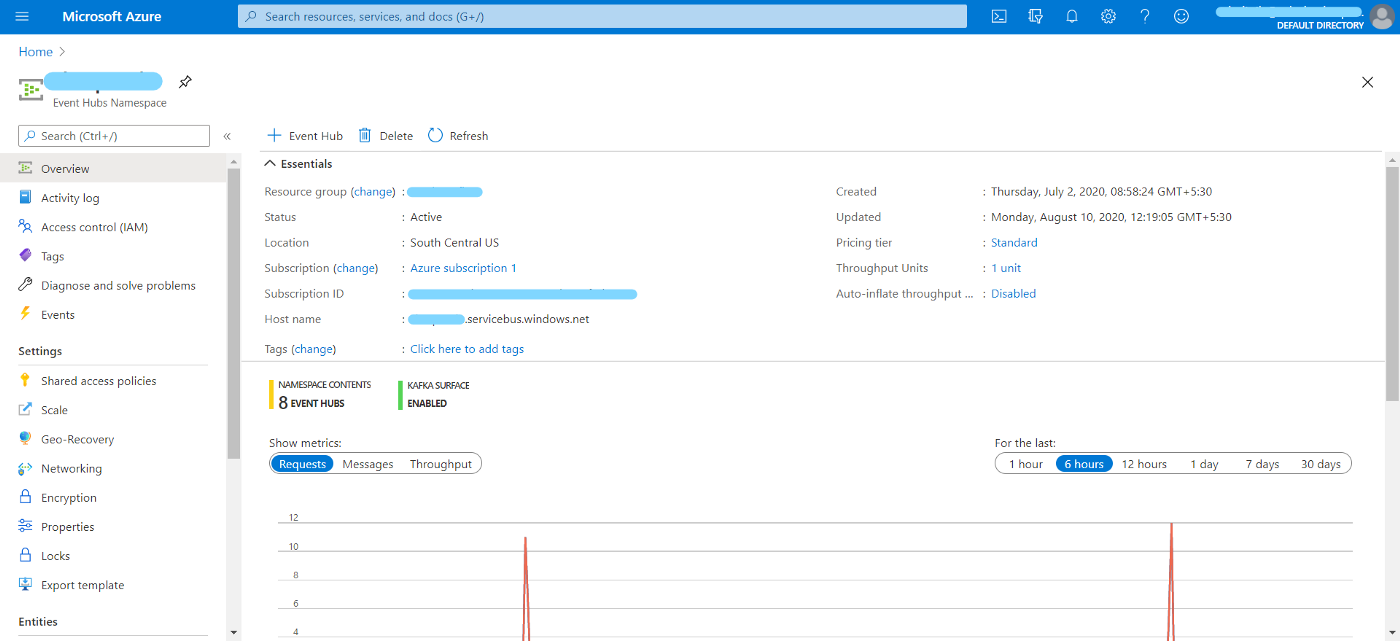
[Download Debezium](https://repo1.maven.org/maven2/io/debezium/debezium-connector-sqlserver/1.3.0.Final/debezium-connector-sqlserver-1.3.0.Final-plugin.tar.gz)  
  
Extract the files from the above download, copy the .jar files from debezium-connector-sqlserver into kafka\kafka\_2.13–2.6.0\libs folder.

**Configure Kafka connect distributed properties:**

Create a file (or replace the file) named connect-distributed.properties inside kafka\kafka\_2.13–2.6.0\config and copy the following contents into that file.

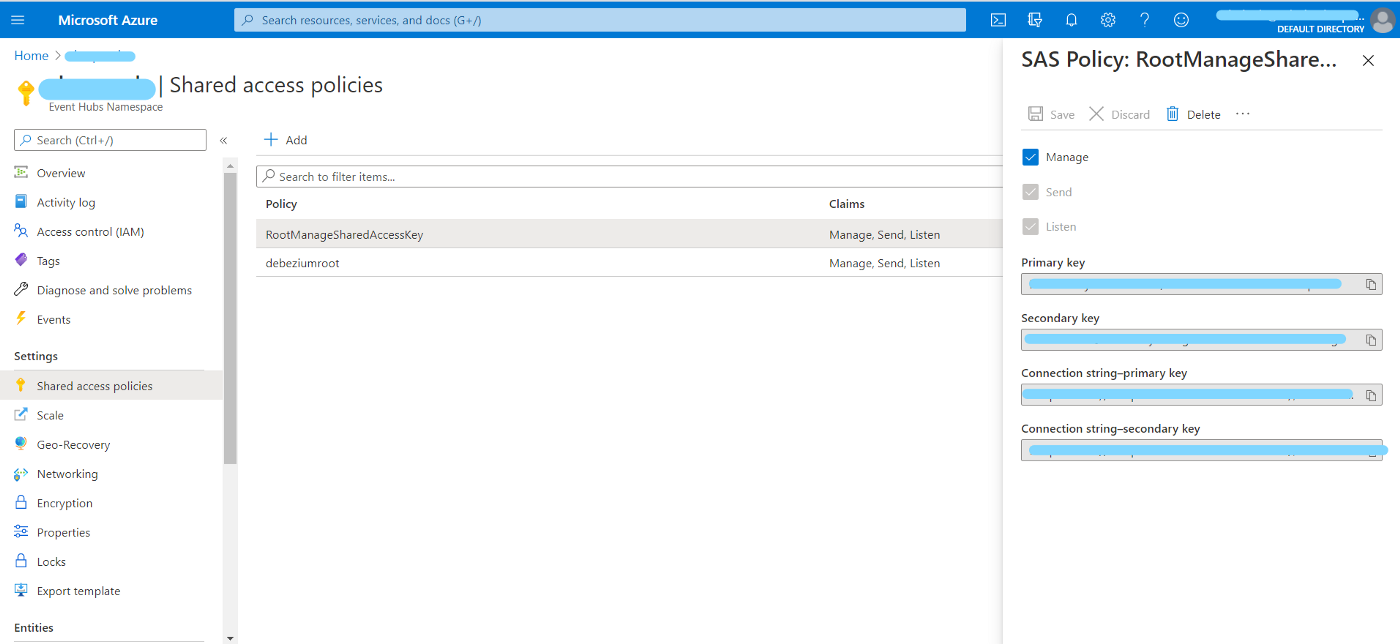
**bootstrap.servers=namespace.servicebus.windows.net:9093  
group.id=connect-cluster-group  
# connect internal topic names, auto-created if not exists  
config.storage.topic=connect-cluster-configs  
offset.storage.topic=connect-cluster-offsets  
status.storage.topic=connect-cluster-status  
# internal topic replication factors - auto 3x replication in Azure Storage  
config.storage.replication.factor=1  
offset.storage.replication.factor=1  
status.storage.replication.factor=1  
rest.advertised.host.name=connect  
offset.flush.interval.ms=10000  
key.converter=org.apache.kafka.connect.json.JsonConverter  
value.converter=org.apache.kafka.connect.json.JsonConverter  
internal.key.converter=org.apache.kafka.connect.json.JsonConverter  
internal.value.converter=org.apache.kafka.connect.json.JsonConverter  
internal.key.converter.schemas.enable=false  
internal.value.converter.schemas.enable=false  
# required EH Kafka security settings  
security.protocol=SASL\_SSL  
sasl.mechanism=PLAIN  
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username= "$ConnectionString" password= "PASSWORD";  
producer.security.protocol=SASL\_SSL  
producer.sasl.mechanism=PLAIN  
producer.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username= "$ConnectionString" password="PASSWORD";  
consumer.security.protocol=SASL\_SSL  
consumer.sasl.mechanism=PLAIN  
consumer.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username= "$ConnectionString" password="PASSWORD";  
plugin.path=E:\kafka\kafka\_2.13–2.6.0\libs # path to the libs directory within the Kafka release**

Copy the Host Name from Azure Event Hub Namespace and copy it to bootstrap.servers value.



Go to Shared access policies, select your policy and copy the Connection string primary key and replace “PASSWORD” with it in connect-distributed.properties. The connection string looks something like:

“Endpoint=sb://namespace.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=mkengtewngwngwk/gkewekbg=”



Also, replace plugins.path value with path to libs folder in kafka.

Save connect-distributed.properties inside config folder.

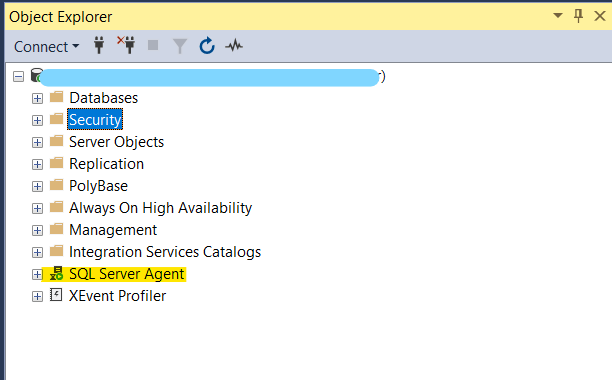
Go to the directory where you have the kafka folder, open Git Bash and run the following command.  
  
**./bin/connect-distributed.sh ./config/connect-distributed.properties**

Now we have kafka connect running. Moving forward, we’ll set-up SQL Server CDC, and configure it to post change events to Event Hub.

**Setting up SSMS and enabling CDC (Change Data Capture) on Table:**

Follow the above [**link**](https://www.microsoft.com/en-gb/sql-server/sql-server-downloads) to download SQL Server 2019 Developer.

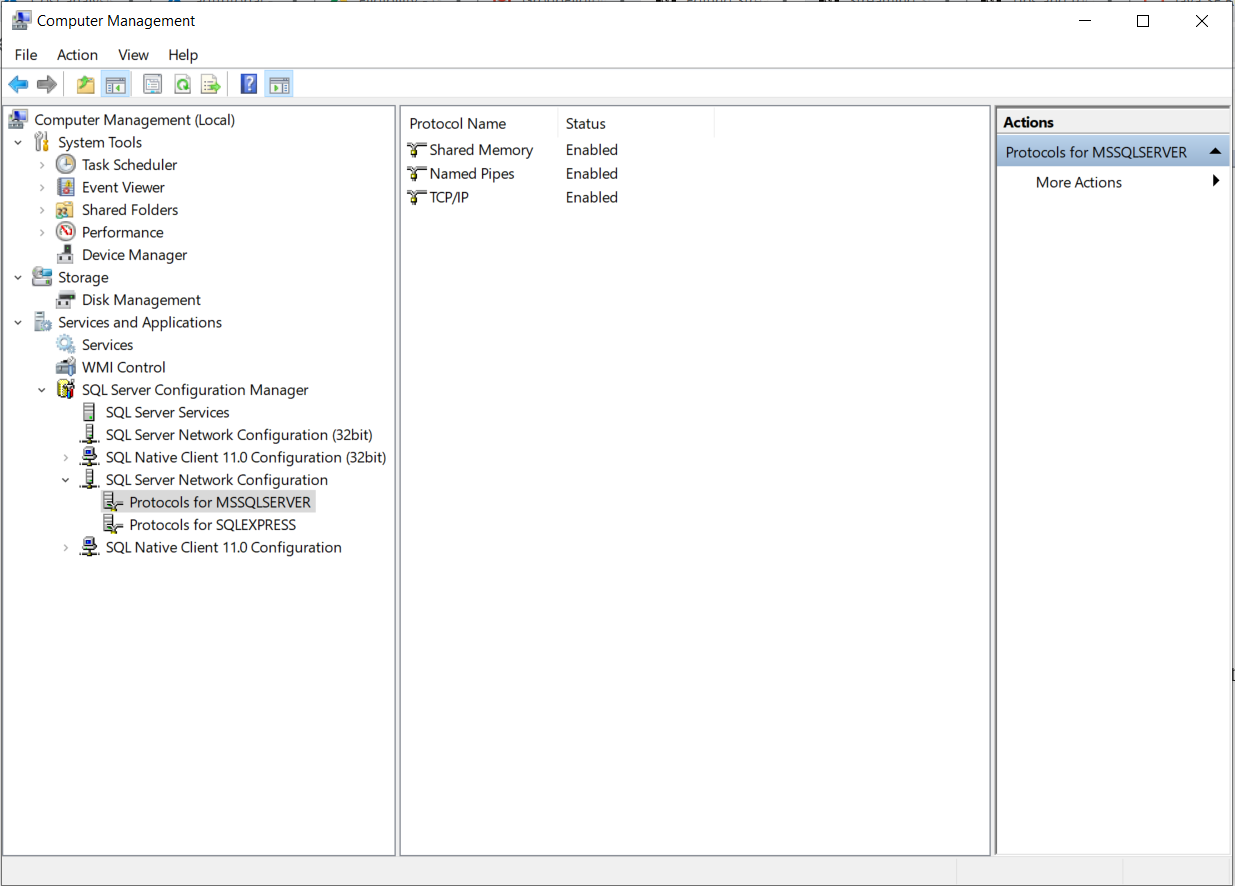
Once you download and install the SSMS, create a login, and provide it ‘**sysAdmin’** role. Make sure that SQL server agent is running. If not, right click on it, and select Start.



Also, assure that you have TCP/IP request enabled. To check for same, press Ctrl+R and run

**compmgmt.msc**

Go to SQL Server Network Configuration in SQL Server Configuration Manager and enable TCP/IP.



**Enabling CDC on table**

Create a database and table and run the below script to enable CDC for your table:

To enable CDC for your database, click new query in SSMS and run:

* **EXEC sys.sp\_cdc\_enable\_db**
* **EXEC sys.sp\_cdc\_enable\_table   
  @source\_schema = N'dbo',   
  @source\_name = N'users',  
  @role\_name = N'MyRole',  
  @supports\_net\_changes = 1**
* **EXEC sys.sp\_cdc\_help\_change\_data\_capture**

‘users’ is the name of the table targeted for CDC.   
‘MyRole’ is the role for which CDC is to be enabled.

Now, create a file named sqlserverproperties.json and copy the following contents to the file.

**{  
"name": "sql-server-connection",   
"config": {  
"connector.class": "io.debezium.connector.sqlserver.SqlServerConnector",   
"database.hostname": "localhost",  
"database.port": "1433",   
"database.user": "<user>",   
"database.password": "<password>",   
"database.dbname": "<cdcdb>",   
"database.server.name": "<servername>",   
"table.whitelist": "dbo.users",  
"database.history":"io.debezium.relational.history.MemoryDatabaseHistory"  
}  
}**

Change database.user, database.password, database.dbname and table.whitelist with your SQL server settings and save the file to kafka folder.

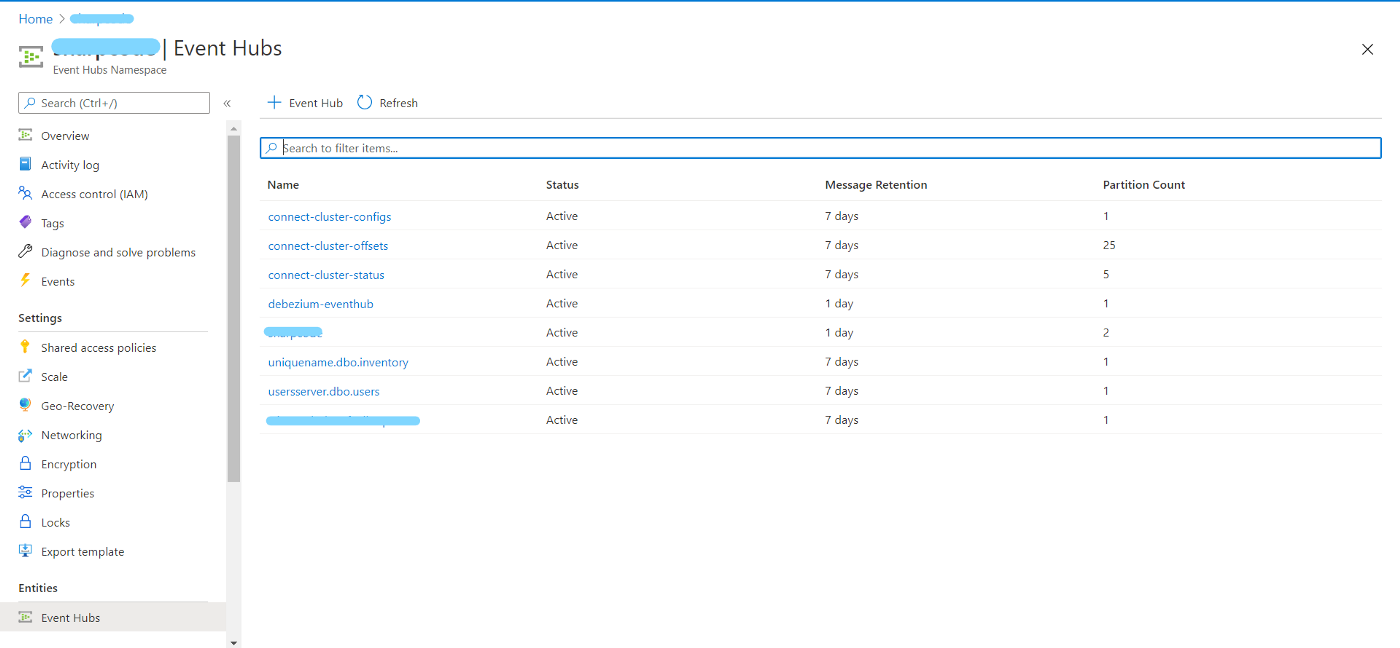
Open bash, and run the following command to post your configurations.

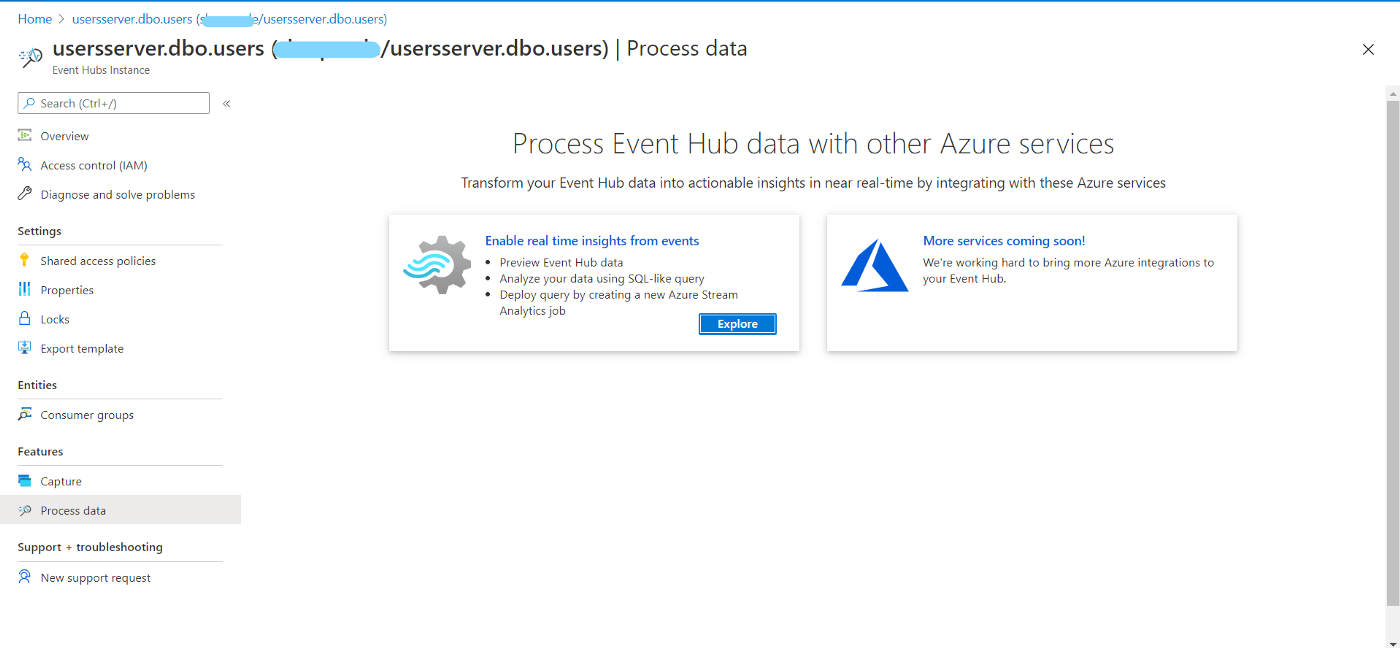
**curl -X POST -H "Accept:application/json" -H “Content-Type: application/json” — data @sqlserverproperties.json http://localhost:8083/connectors**

To check for status, run,

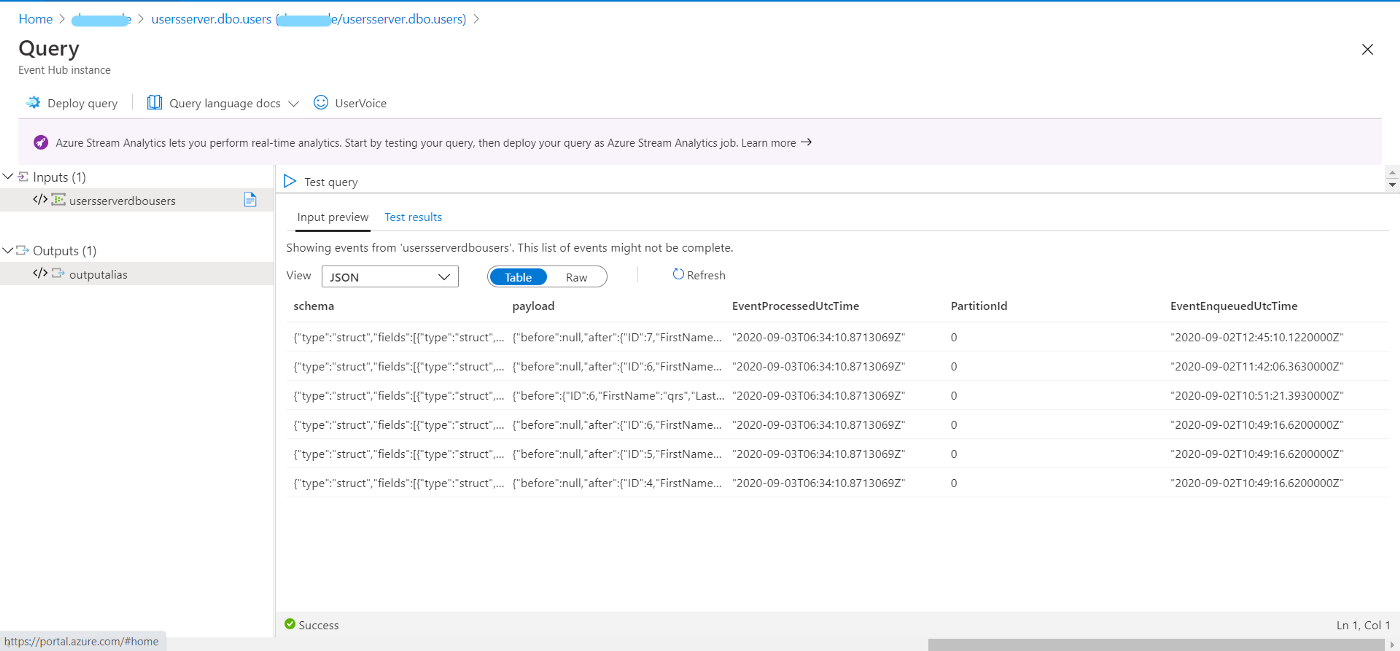
**curl -s http://localhost:8083/connectors/sql-server-connection/status**

Now that you have successfully setup everything, let’s make some changes to the table and look at the logs in Azure Event Hub. Go to Event Hubs, select **<servername>.dbo.users**, Process Data and Explore.





You can see the change events captured from your table in the input preview.



And, IT’S DONE!!!

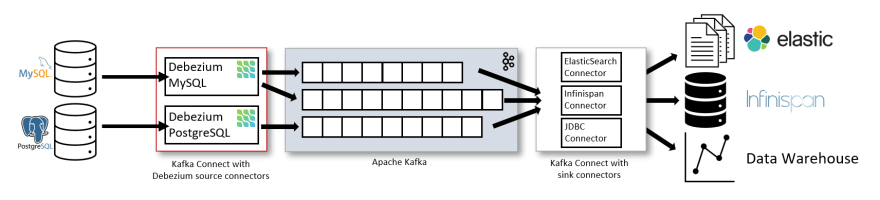
**3rd POC: Set up a Change Data Capture architecture on Azure using Debezium, Postgres and Kafka.**

*Change Data Capture (CDC)* is a technique used to track row-level changes in database tables in response to create, update and delete operations. Different databases use different techniques to expose these change data events - for example, [logical decoding in PostgreSQL](https://www.postgresql.org/docs/current/static/logicaldecoding-explanation.html), [MySQL binary log (binlog)](https://dev.mysql.com/doc/internals/en/binary-log-overview.html) etc. This is a powerful capability, but useful only if there is a way to tap into these event logs and make it available to other services which depend on that information.

[Debezium](https://debezium.io/) does just that! It is a distributed platform that builds on top of Change Data Capture features available in different databases. It provides a set of [Kafka Connect connectors](https://debezium.io/documentation/reference/1.2/connectors/index.html) which tap into row-level changes (using CDC) in database table(s) and convert them into event streams. These event streams are sent to [Apache Kafka](https://kafka.apache.org/) which is a scalable event streaming platform - a perfect fit! Once the change log events are in Kafka, they will be available to all the downstream applications.

This is different compared to the "polling" technique adopted by the [Kafka Connect JDBC connector](https://github.com/confluentinc/kafka-connect-jdbc)

The diagram (from the debezium.io website) summarises it nicely!



This part of article is a guide to getting started with setting up a change data capture based system on Azure using Debezium, [Azure DB for PostgreSQL](https://docs.microsoft.com/azure/postgresql/overview?WT.mc_id=devto-blog-abhishgu) and [Azure Event Hubs](https://docs.microsoft.com/azure/event-hubs/event-hubs-about?WT.mc_id=devto-blog-abhishgu) (for Kafka). It will use the [Debezium PostgreSQL connector](https://debezium.io/documentation/reference/1.2/connectors/postgresql.html) to stream database modifications from PostgreSQL to Kafka topics in Azure Event Hubs

The related config files are available in the GitHub repo <https://github.com/abhirockzz/debezium-azure-postgres-cdc>

Although I have used managed Azure services for demonstration purposes these instructions should work for any other setup as well e.g. a local Kafka cluster and PostgreSQL instance.

**Setup PostgreSQL and Kafka on Azure**

This section will provide pointers on how to configure Azure Event Hubs and Azure DB for PostgreSQL.

Azure DB for PostgreSQL is a managed, relational database service based on the community version of open-source PostgreSQL database engine, and is available in two deployment modes.

You can setup PostgreSQL on Azure using a variety of options including, the [Azure Portal](https://docs.microsoft.com/azure/postgresql/quickstart-create-server-database-portal?WT.mc_id=devto-blog-abhishgu), [Azure CLI](https://docs.microsoft.com/azure/postgresql/quickstart-create-server-database-azure-cli?WT.mc_id=devto-blog-abhishgu), [Azure PowerShell](https://docs.microsoft.com/azure/postgresql/quickstart-create-postgresql-server-database-using-azure-powershell?WT.mc_id=devto-blog-abhishgu), [ARM template](https://docs.microsoft.com/azure/postgresql/quickstart-create-postgresql-server-database-using-arm-template?tabs=azure-portal&WT.mc_id=devto-blog-abhishgu).

Please ensure that you keep the following PostgreSQL related information handy since you will need them to configure the Debezium Connector in the subsequent sections - database hostname (and port), username, password.

**Azure Event Hubs**

Azure Event Hubs is a fully managed data streaming platform and event ingestion service. [It also provides a Kafka endpoint](https://docs.microsoft.com/azure/event-hubs/event-hubs-for-kafka-ecosystem-overview?WT.mc_id=devto-blog-abhishgu) that supports Apache Kafka protocol 1.0 and later and works with existing Kafka client applications and other tools in the Kafka ecosystem including Kafka Connect.

After the setup, please ensure that you keep the Connection String handy since you will need it to configure Kafka Connect. You can do so [using the Azure Portal](https://docs.microsoft.com/azure/event-hubs/event-hubs-get-connection-string?WT.mc_id=devto-blog-abhishgu) or [Azure CLI](https://docs.microsoft.com/cli/azure/eventhubs/eventhub/authorization-rule/keys?view=azure-cli-latest&WT.mc_id=devto-blog-abhishgu#az-eventhubs-eventhub-authorization-rule-keys-list).

**Install Kafka**

To run Kafka Connect, I will be using a local Kafka installation just for convenience. [**Just download Apache Kafka**](https://kafka.apache.org/downloads), unzip its contents and you're good to go!

**[Download Debezium connector](https://repo1.maven.org/maven2/io/debezium/debezium-connector-postgres/1.3.0.Final/debezium-connector-postgres-1.3.0.Final-plugin.tar.gz) and start Kafka Connect**

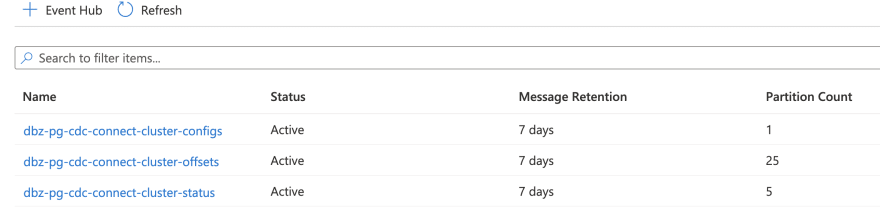
Extract the files from the above download, copy the .jar files from debezium-connector-postgres into kafka\kafka\_2.13–2.6.0\libs folder.  
Also, replace plugins.path value with path to libs folder in kafka.

Save connect.properties inside config folder.

Go to the directory where you have the kafka folder, open Git Bash and run the following command.

**./bin/connect-distributed.sh ./config/connect.properties**

Wait for the Kafka Connect instance to start - you should see Kafka Connect internal topics in Azure Event Hubs e.g.



**Configure PostgreSQL:**Follow the steps [mentioned here](https://docs.microsoft.com/en-us/azure/postgresql/quickstart-create-server-database-portal) for configuring the Azure Database PostgreSQL Server.

Before installing the connector, we need to:

* Ensure that the PostgreSQL instance is accessible from your Kafka Connect cluster.
* Ensure that the PostgreSQL replication setting is set to "Logical". Go to the Replication in Settings on the Azure Database PostgreSQL Server and set it to “Logical”.
* Create a table which you can use to try out the change data capture feature.

If you're using Azure DB for PostgreSQL, create a firewall rule using either [az postgres server firewall-rule create](https://docs.microsoft.com/en-us/azure/postgresql/howto-manage-firewall-using-cli" \l "create-firewall-rule) command or simple navigate to Connection Security to whitelist your Kafka Connect host.

In our case, it is a local Kafka Connect cluster, so we navigate to the Azure portal (**Connection security** section of my PostgreSQL instance) and click **Add current client IP address** to make sure that local IP is added to the firewall rule.

After updating the configuration, you will need to re-start the server. Once the database is up and running, create the table - I have used psql CLI in this example, but feel free to use any other tool. For example, to connect to your PostgreSQL database on Azure over SSL (you will be prompted for the password).

Run the commands below in the psql CLI:  
  
**psql -h <POSTGRESQL\_INSTANCE\_NAME>.postgres.database.azure.com -p 5432 -U <POSTGRES\_USER\_NAME> -W -d <POSTGRES\_DB\_NAME> --set=sslmode=require**

Replace <POSTGRESQL\_INSTANCE\_NAME>, <POSTGRES\_USER\_NAME> and <POSTGRES\_DB\_NAME> with your server credentials.

//Create the table

**CREATE TABLE todos (id SERIAL, description VARCHAR(30), todo\_status VARCHAR(10), PRIMARY KEY(id));**

**Install Debezium PostgreSQL source connector:**

Update the “pg-source-connector.json” file with the details for the Azure PostgreSQL instance as below.  
**{**

**"name": "<pg-connector>",**

**"config": {**

**"connector.class":io.debezium.connector.postgresql.PostgresConnector",**

**"database.hostname": "<PostgresInstanceName>.postgres.database.azure.com",**

**"database.port": "5432",**

**"database.user": "<DB\_UserName>",**

**"database.password": "<DB\_Password>",**

**"database.dbname": "<DB\_Name e.g. postgres>",**

**"database.server.name": "<Logical\_Namespace e.g. todo-server>",**

**"plugin.name": "wal2json",**

**"table.whitelist": "<Table\_Names e.g. public.todos>"**

**}**

**}**Let's go through the configuration:

For detailed info, check [Debezium documentation](https://debezium.io/documentation/reference/1.2/connectors/postgresql.html#postgresql-connector-properties)

* connector.class: name of the connector class (this is a static value)
* database.hostname and database.port: IP address or hostname for your PostgreSQL instance as well as the port (e.g. 5432)
* database.user and database.password: username and password for your PostgreSQL instance
* database.dbname: database name e.g. postgres
* database.server.name: Logical name that identifies and provides a namespace for the particular PostgreSQL database server/cluster being monitored.
* table.whitelist: comma-separated list of regex specifying which tables you want to monitor for change data capture
* plugin.name: name of the logical decoding plug-in e.g. wal2json

At the time of writing, Debezium supports the following plugins: decoderbufs, wal2json, wal2json\_rds, wal2json\_streaming, wal2json\_rds\_streaming and pgoutput. I have used wal2json in this example, and it's [supported on Azure as well](https://docs.microsoft.com/azure/postgresql/concepts-logical?WT.mc_id=devto-blog-abhishgu)!  
  
**Finally, creating the connector!  
curl -X POST -H "Accept:application/json" -H "Content-Type: application/json" --data @pg-source-connector.json http://localhost:8083/connectors**

Change data capture in action!!! Run the commands below in psql CLI:

**psql -h <POSTGRES\_INSTANCE\_NAME>.postgres.database.azure.com -p 5432 -U <POSTGRES\_USER\_NAME> -W -d <POSTGRES\_DB\_NAME> --set=sslmode=require**

**INSERT INTO todos (description, todo\_status) VALUES ('install postgresql', 'complete');**

**INSERT INTO todos (description, todo\_status) VALUES ('install kafka', 'complete');**

**INSERT INTO todos (description, todo\_status) VALUES ('setup source connector', 'pending');**

**The connector should now spring into action and send the CDC events to an Event Hubs topic named <server name in config>.<table name> e.g. todo-server.public.todos  
  
Now follow the steps as mentioned in the 2nd POC:**

* Go to the Event Hub Topic created (e.g.: todo-server.public.todos).
* Click on “**Process Data** in the Feature Menu”.
* Click **Explore** in the “Enable Real Time Insights From Events”.
* Click Test Query and see the results.

And, it’s done!!!

Steps further:

* A Web/Console Application or similar for doing the CRUD operations in the database and create CDC events.
* These CDC events would be captured by Debezium and will be reflected in the AEH.
* After above steps, you can deploy the query after clicking Explore button in the Process Data and create a stream analytics job.

You can Output the events data in any Storage or Database or Azure Function or similar. Check out for [**Stream Analytics Jobs**](https://docs.microsoft.com/en-us/azure/event-hubs/process-data-azure-stream-analytics)**.**

**References:**

* <https://vladmihalcea.com/how-to-extract-change-data-events-from-mysql-to-kafka-using-debezium/>
* <https://medium.com/@reveation.labs/streaming-events-from-sql-server-to-event-hub-in-azure-using-debezium-55dfd1a0e214>
* <https://dev.to/azure/tutorial-set-up-a-change-data-capture-architecture-on-azure-using-debezium-postgres-and-kafka-49h6>
* <https://docs.microsoft.com/en-us/azure/event-hubs/process-data-azure-stream-analytics>
* <https://debezium.io/documentation/reference/1.3/tutorial.html>
* <https://kafka.apache.org/documentation/>
* <https://sookocheff.com/post/kafka/kafka-in-a-nutshell/#:~:text=Kafka%20topics%20are%20divided%20into,from%20a%20topic%20in%20parallel>

**CHEERS!!!**