PostGIS – Data Audit Triggers

In this blog we will explore **adding triggers** into a **PostGIS database** to monitor the changes being made to a PostGIS table via QGIS.

This blog is inspired by the commonly asked question at many of my pre-sales meetings, where I am illustrating shared project working - integrating Open Source GIS and Autodesk BIM applications with one source of truth via a PostGIS database…..

Q – Are you able to track the data changes made in QGIS, webGIS, Infraworks, Map3D?

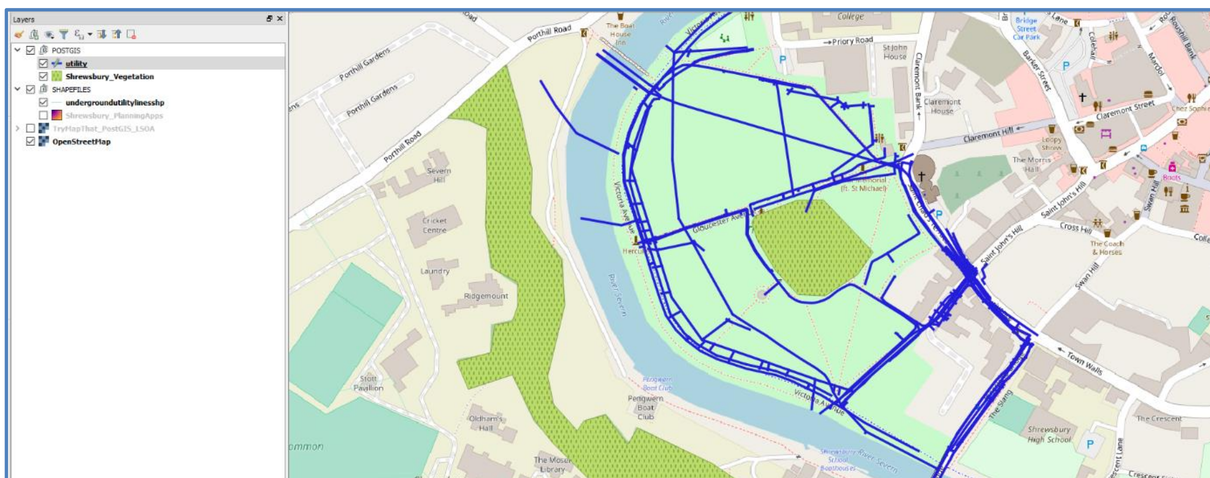
…. and while this wasn't something that I had implemented before myself, I understood its importance to Data Controllers, who need to understand how their assets have evolved and what changes have occurred to environmental, social, and economic datasets within their projects.

So, my answer has always been **yes** of course it can! it's a database and you can do anything in a database 😉

However, I thought it was time to set aside a little bit of RnD and prove this to myself and develop a live working example to really demonstrate the power of storing your assets and spatial information within a spatial database, such as PostGIS.

Shrewsbury Utility Data:

In this example, we are using a **Utility** Pipeline dataset in Shrewsbury, England.



These assets were formerly locked down within Autodesk Map3D, until we undertook a **MAPEXPORT** command and created a Shapefile. We then uploaded that Shapefile into a PostGIS database, on our public facing TryMapThat server. This created a table called **'utility'**, which had geometry, a layer type, height value and a unique id.

Data Output | Explain | Messages | Query History

	geom geometry	layer character varying (254)	height numeric	fid [PK] character varying (100)
432	010500002...	LINE_ELEC	5	{f209d657-3227-4602-bb4a-cb3a...
433	010500002...	LINE_ELEC	5	{f303e823-2baa-4f76-aadc-90cb...
434	010500002...	LINE_ELEC	5	{f30c7ff7-60ab-4952-814a-f0bdf9...
435	010500002...	LINE_SWS	5	{f33b4857-05ef-43b4-bad1-0b62...
436	010500002...	LINE_CCTV	5	{f3534a4b-3ec2-497e-a571-8537...
437	010500002...	LINE_ELEC	5	{f4d738ce-e110-4994-a80b-023e...
438	010500002...	LINE_CCTV	5	{f544de05-80f2-46fe-8695-d444c...
439	010500002...	LINE_ELEC	5	{f5857e4f-c5f6-47f8-b600-96958...

I utilise this spatial data to demonstrate an example of **shared project working**, where the same dataset is being accessed, and indeed edited, via several applications, including; QGIS, MapThat webGIS, Map3D and Infraworks. As one application makes a change to the underlying data, all users will see those changes live, without the need to export, translate and reimport datasets. Feel free to view that blog using the link below:

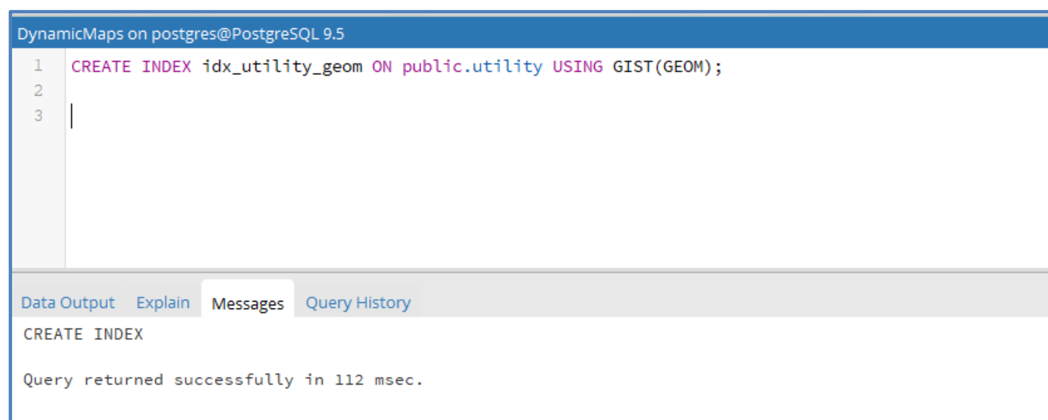
https://www.cadlinecommunity.co.uk/hc/en-us/articles/115003797389-Shared-Project-Working-Implementing-GIS-Interoperability

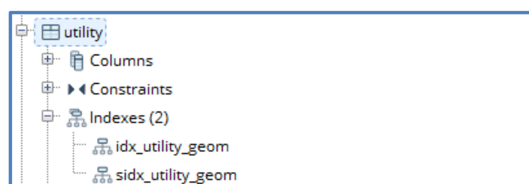
So, based on my pre-sales feedback and for my own curiosity I decided to research how simple it is to create database **triggers in PostGIS** to record these data changes. Below I will describe the steps which I undertook to create database triggers so that the next time that the data was updated we can track those changes and provide an **Audit history.**

Step 1 - Index Existing Table:

Firstly, ensure that your spatial table has an index. The table that we are using is a pipeline table called **utility.**

CREATE INDEX idx_utility_geom ON public.utility USING GIST(GEOM);





Step 2 – Copy the Spatial Table (utility) to create a History Table (utility_history):

Next, we will need a database table to write the data change records into. The simplest way to do this is to create a copy of the existing spatial table:

CREATE TABLE utility_history AS

TABLE utility;

And then to delete all the existing records from the new table:

TRUNCATE utility_history;

DELETE FROM utility_history;

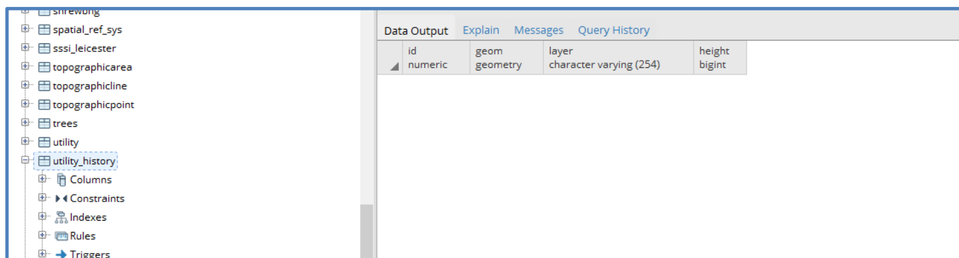
DynamicMaps on postgres@PostgreSQL 9.5

```
1  TRUNCATE utility_history;
2  DELETE FROM utility_history;
```

Data Output | Explain | Messages | Query History

```
DELETE 0

Query returned successfully in 90 msec.
```



Step 3– Alter the Schema of the History Table:

Next, we will need to alter the design of the new utility_history table to add some extra fields which will be used to **record the changes** we make. Into the Utility History table we will add a created timestamp, created_by, deleted timestamp, deleted_by, updated_timestamp and updated_by.

ALTER TABLE utility_history ADD COLUMN created timestamp;

ALTER TABLE utility_history ADD COLUMN created_by varchar(50);

ALTER TABLE utility_history ADD COLUMN deleted timestamp;

ALTER TABLE utility_history ADD COLUMN deleted_by varchar(50);

ALTER TABLE utility_history ADD COLUMN updated timestamp;

ALTER TABLE utility_history ADD COLUMN updated_by varchar(50);

ALTER TABLE utility_history ADD COLUMN modified boolean;

Data Output | Explain | Messages | Query History

	id numeric	geom geometry	layer character varying (254)	height bigint	created timestamp without time zone	created_by character varying (50)	deleted timestamp without time zone	deleted_by character varying (50)	modified boolean
1	105	010500002...	LINE_GSV	5	[null]	[null]	[null]	[null]	[null]
2	249	010500002...	LINE_UNID	5	[null]	[null]	[null]	[null]	[null]
3	39	010500002...	LINE_WATER	5	[null]	[null]	[null]	[null]	[null]
4	11	010500002...	LINE_GAS	5	[null]	[null]	[null]	[null]	[null]
5	13	010500002...	LINE_DUCT	5	[null]	[null]	[null]	[null]	[null]
6	14	010500002...	LINE_ELEC	5	[null]	[null]	[null]	[null]	[null]
7	15	010500002...	LINE_GSV	5	[null]	[null]	[null]	[null]	[null]
8	16	010500002...	LINE_GSV	5	[null]	[null]	[null]	[null]	[null]
9	12	010500002...	LINE_DATA	5	[null]	[null]	[null]	[null]	[null]

Step 4– Alter the Schema of the History Table:

In this step we need to **add an Index** to the updated utility_history table.

CREATE INDEX idx_utility_history_geom ON public.utility_history USING GIST(GEOM);

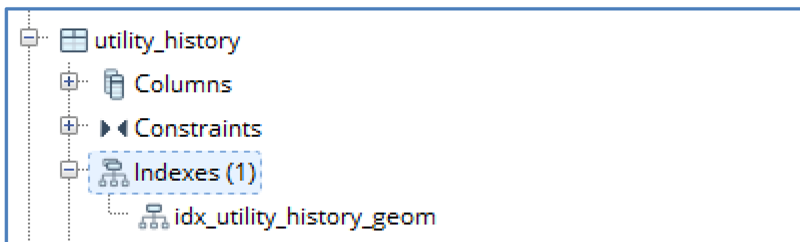
DynamicMaps on postgres@PostgreSQL 9.5

```
1  CREATE INDEX idx_utility_history_geom ON public.utility_history USING GIST(GEOM);
2
3
4
```

Data Output | Explain | Messages | Query History

```
CREATE INDEX

Query returned successfully in 81 msec.
```



Step 5 – Create a Database Trigger (*Utility_history_tracker)*:

Now we can create some triggers in the database. The **first Trigger** that we create will be at **database level**. It will run anytime that PostGIS sees that there has been a change to the Utility Table, and uses a series of Else If queries to understand if the change is an Update, Insert or Deletion. It is used to insert into these a record of changes into the utility_history table.

```
CREATE OR REPLACE FUNCTION public.utility_history_tracker()
RETURNS trigger AS
$utility_history_tracker$
 BEGIN
    IF (TG_OP = 'INSERT') THEN
      INSERT INTO public.utility_history
        (geom, layer, height, fid, created, created_by, modified)
      VALUES
        (NEW.geom, NEW.layer, NEW.height, NEW.fid, current_timestamp, current_user, FALSE);
      RETURN NEW;
    ELSIF (TG_OP = 'UPDATE') THEN
      UPDATE public.utility_history
        SET updated = current_timestamp, updated_by = current_user, modified = TRUE
        WHERE deleted IS NULL and fid = OLD.fid;
      INSERT INTO public.utility_history
        (geom, layer, height, fid, updated, updated_by, modified)
      VALUES
        (NEW.geom, NEW.layer, NEW.height, NEW.fid, current_timestamp, current_user, FALSE);
      RETURN NEW;
     ELSIF (TG_OP = 'DELETE') THEN
      UPDATE public.utility_history
        SET deleted = current_timestamp, deleted_by = current_user
        WHERE deleted IS NULL and fid = OLD.fid;
```

RETURN NULL;

END IF;

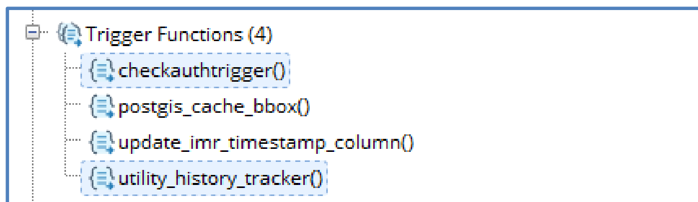
END;

$utility_history_tracker$

LANGUAGE plpgsql VOLATILE

COST 100;

Once the SQl script has been ran, you will now have created a Database level trigger.



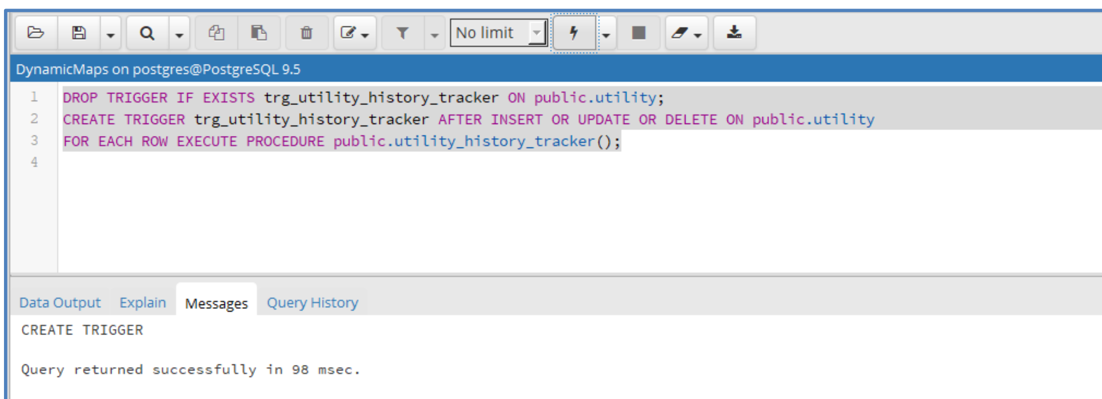
Step 6 – Create a Table Trigger (*trg_utility_history_tracker)*:

The second Trigger that we create will be for the Utility Table. This trigger will check for any Insertions, Deletions or Updates into the Utility Table, and when a change is made it will execute the ***Utility_History_Tracker*** *trigger* at Database level to update the history table.

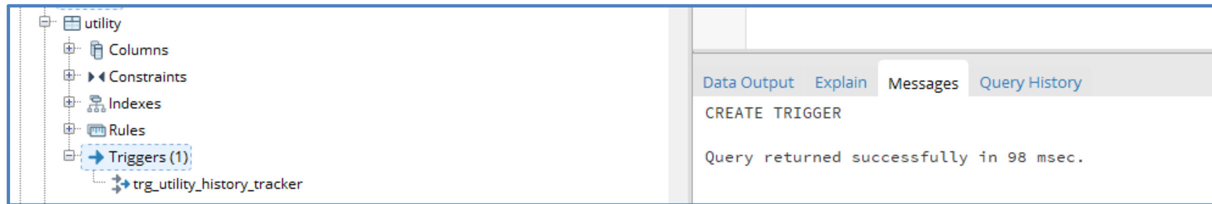
DROP TRIGGER IF EXISTS trg_utility_history_tracker ON public.utility;

CREATE TRIGGER trg_utility_history_tracker AFTER INSERT OR UPDATE OR DELETE ON public.utility

FOR EACH ROW EXECUTE PROCEDURE public.utility_history_tracker();



Once the SQl script has been ran, you will now have created a **Table level** trigger in the Utility table.



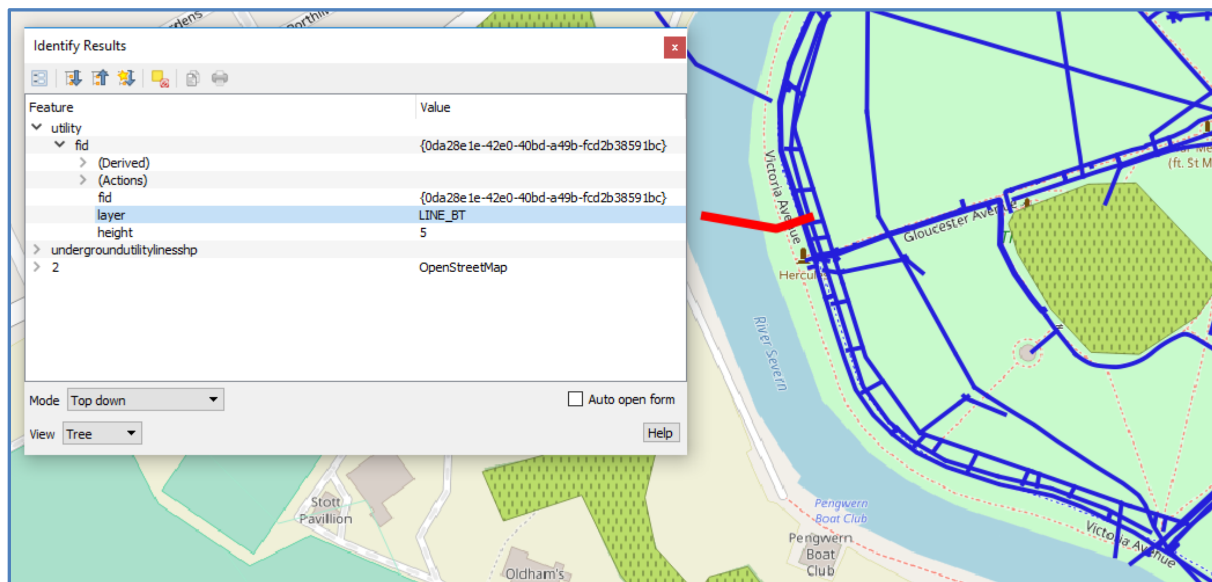
Step 7 – Test that the Triggers work:

To test to see if the Triggers are successfully recording the data changes we will make **3 different** types of edit.

(a) Attribute Edit:

Using the **PostGIS SQL Tool**, we can use a script to update an existing record and change the layer type value.

Currently the pipeline with id *{0da28e1e-42e0-40bd-a49b-fcd2b38591bc}* is of type **LINE_BT.**



If we now run the following SQL script within PostGIS:

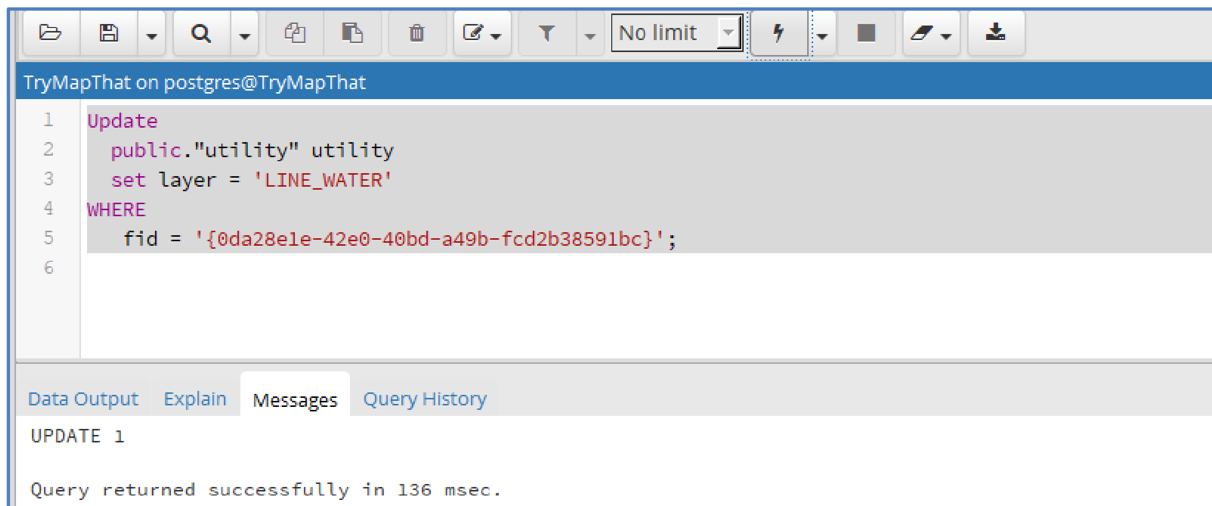
Update

public."utility" utility

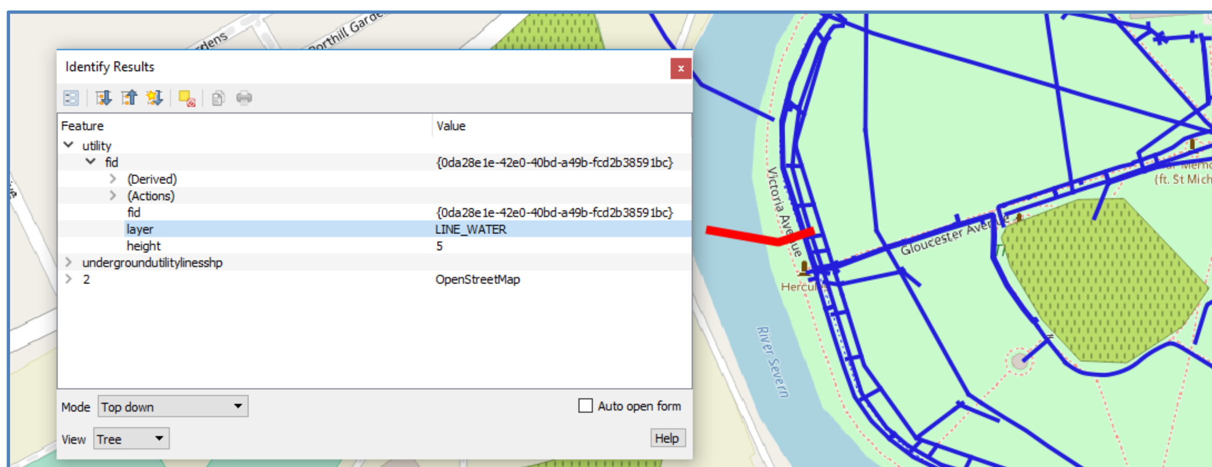
set layer = 'LINE_WATER'

WHERE

fid = '{0da28e1e-42e0-40bd-a49b-fcd2b38591bc}';



the same record will be updated to now be **LINE_WATER.**



If we now check the **utility_history** table we can see that a change log record has been added, which records that for this record ID, there has been an update of the layer field to be LINE_WATER, and it also records the date/time and user that made the update (in this case postgres).

```
1 SELECT * FROM public.utility_history
2
```

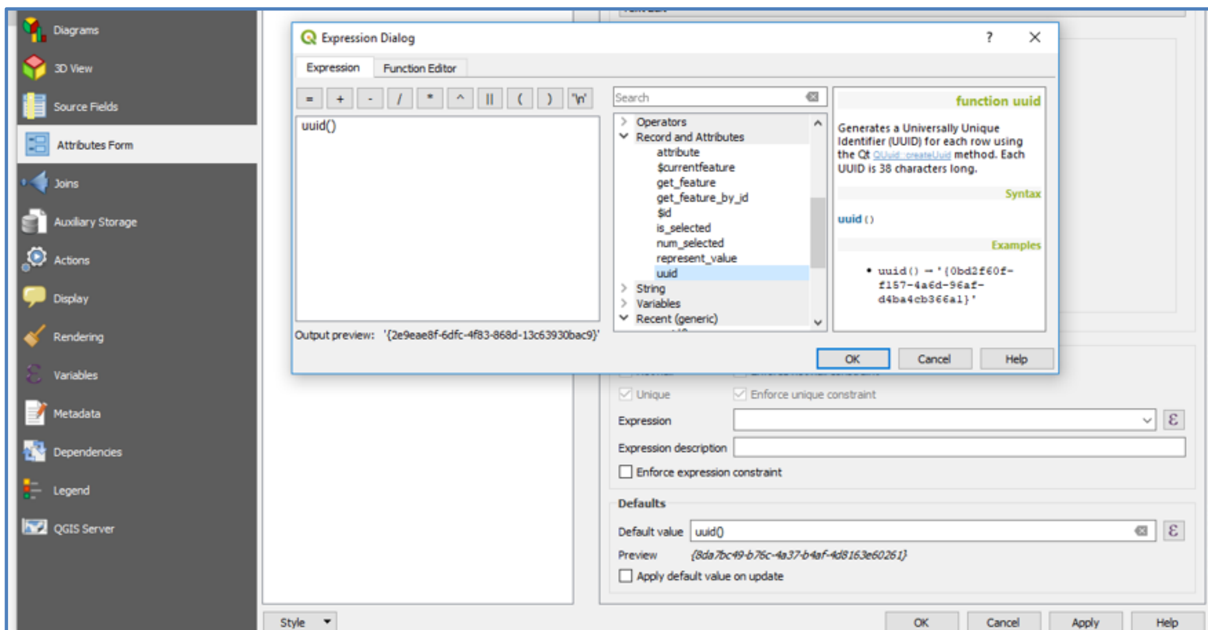
Data Output | Explain | Messages | Query History

	fid character varying	geom geometry	layer character varying (254)	height bigint	updated timestamp without time zone	updated_by character varying (50)
1	{0da28e1e-42e0-40b...	010500002...	LINE_WATER	5	2018-03-19 16:39:41.890168	postgres

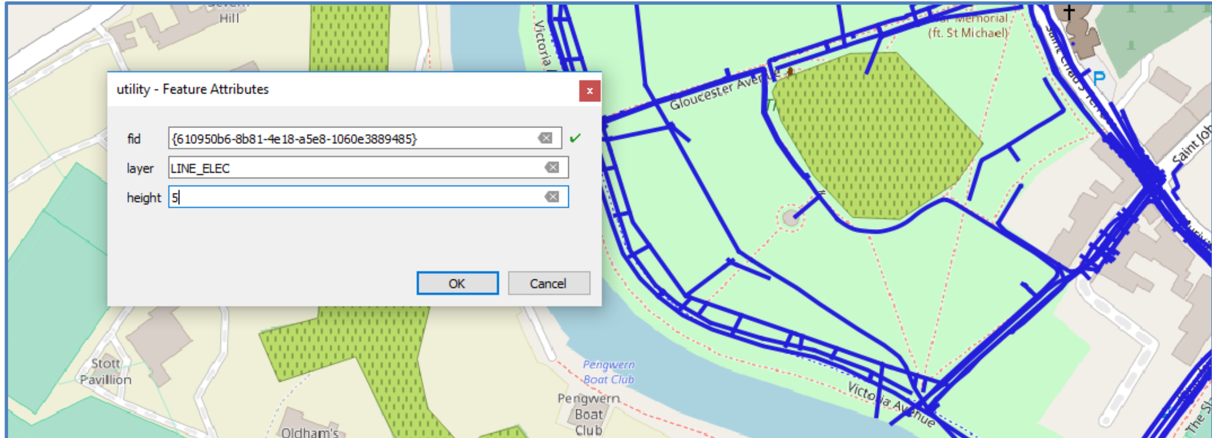
Insert New Records:

Using QGIS, we can **add a new feature** and see that change to the utility table added into the utility_history table as our triggers create an audit history automatically.

Tip – when inserting records via QGIS you can use the layer properties – **Attributes Form** – to apply a **Unique Identifier** as a default value when updates are made to the database table!



This means that once we have **digitised a new pipeline** into the utility table, the attribute form will automatically create a **unique id** for the FID value.



Once that insertion has been **saved in QGIS**, if you now return to PostGIS and refresh the utility_history table, we can see that a new record has been added, and the **created** and **created_by** fields have been updated.

```
1  SELECT * FROM public.utility_history
2
```

Data Output | Explain | Messages | Query History

	fid character varying	geom geometry	layer character varying (254)	height bigint	created timestamp without time zone	created_by character varying (50)	deleted timestamp without time zone	deleted_by character varying
1	{0da28e1e-42e0-40b...	010500002...	LINE_WATER	5	[null]	[null]	[null]	[null]
2	{610950b6-8b81-4e1...	010500002...	LINE_ELEC	5	2018-03-19 16:15:18.759545	postgres	[null]	[null]

Delete Records:

Finally, we will **delete a record** and check that we are tracking this change via our triggers into the utility_history table. We can either make the deletion via QGIS, or simply run the SQL command below, using the correct ID as required.

DELETE FROM public."utility"

WHERE fid = '{610950b6-8b81-4e18-a5e8-1060e3889485}';

```
1  DELETE FROM  public."utility"
2  WHERE fid = '{610950b6-8b81-4e18-a5e8-1060e3889485}';
3
```

Data Output   Explain   Messages   Query History

```
DELETE 1

Query returned successfully in 94 msec.
```

Checking the **utility_history** table we can see that the change has been recorded, where the **deleted** and **deleted_by** values have been updated to reflect the change to that record.

Data Output   Explain   Messages   Query History

	fid character varying	geom geometry	layer character varying (	height bigint	created timestamp with	created_by character varyi	deleted timestamp without time zone	deleted_by character varying (50)	updated timestamp without t
1	{0da28e1e-42e0-40b...	010500002...	LINE_WATER	5	[null]	[null]	[null]	[null]	2018-03-19 16:07:1(
2	{610950b6-8b81-4e1...	010500002...	LINE_ELEC	5	2018-03-19 ...	postgres	2018-03-19 16:19:31.199902	postgres	[null]

However, currently the triggers I am using will only show a deletion for a record that has already been inserted into the history table – via an Update or an Insert.

…..,so my next RnD will be to see if I can capture deletions on previously non edited records.!

Many thanks to the following links which helped me research creating triggers in PostGIS:

- https://geosymp.com/tracking-history-in-postgis-databases-with-triggers/
- https://stackoverflow.com/questions/12577004/what-does-language-plpgsql-volatile-mean